JPL PUBLICATION 81-67

Proceedings

Conference on the Computing Environment for Mathematical Software

Cosponsored by
JPL and ACM-SIGNUM

Huntington-Sheraton Hotel
Pasadena, California
July 29-31, 1981

July 15, 1981

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL PUBLICATION 81-37

Proceedings

Conference on the Computing Environment for Mathematical Software

Cosponsored by
JPL and ACM-SIGNUM

Huntington-Sheraton Hotel
Pasadena, California
July 29-31, 1981

July 15, 1981

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

This publication was prepared by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

CONTENTS

CONTENTS (cont.)

ABSTRACT

The conference on the Computing Environment for Mathematical Software, held in Pasadena, California on July 29-31, 1981, was cosponsored by the Jet Propulsion Laboratory and the Special Interest Group on Numerical Mathematics of the Association for Computing Machinery (ACM-SIGNUM). The conference provided a sequel to two previous SIGNUM conferences held in Pasadena in 1974 and 1978. Topcis included software tools, Fortran standards activity, and features of languages, operating systems, and hardware that are important for the development, testing, and maintenance of mathematical software. This publication includes extended abstracts of the papers presented at the conference.

PROGRAM

1981 JPL/ACM - SIGNUM

The Computing Environment for Mathematical Software

Huntington Sheraton Hotel, Pasadena, California

Tuesday, July 28

7:00 - 9:00 pm	REGISTRATION

Wednesday, July 29

8:00 - 9:00 am	REGISTRATION
8:55 - 9:00	Welcome, Fred Krogh
	SESSION CHAIRMAN: Lloyd Fosdick
9:00 - 9:20	**Adams, Jeanne, "Fortran Standards, An Overview"
9:20 - 10:00	**Meissner, Loren P., "Core-and-Modules Design for Next Fortran Standard"
10:00 - 10:15	Adams, Jeanne, Chairperson, "Fortran Standards - Question Period"
10:15 - 10:30	BREAK
10:30 - 11:00	**Wilson, Alan, "Parallel Processing in Fortran"
11:00 - 11:30	**Wilkens, Edward J., "Global Data Sharing and Internal Procedures in Fortran 8X"
11:30 - 1:30 pm	BREAK
	SESSION CHAIRMAN: Stan Brown
1:30 - 2:00	**Smith, Brian, "General Precision Data Type Facility and Environmental Inquiry Feature"
2:00 - 2:30	Adams, Jeanne, Chairperson, "Fortran Standards - Question and Discussion Period"
2:30 - 3:05	Miller, Webb, "A Consumer's Report on Fortran Programming Environments"
3:05 - 3:20	BREAK
3:20 - 3:45	Lucas, L. W., "Numerical Software - View from the Trenches"
3:45 - 4:20	**Ryan, Thomas A., "Packaging Statistical Software"
4:20 - 4:45	*Amano, Kaname; Chiba, Masaki; Mochida, Akeno; and Maeda, Takashi, "Algorithm Bank: Information System for Mathematical Software"
4:45 - 5:05	Gaffney, P. W., "Software Management"

* Speaker in case of two or more authors

** Invited Speaker

Thursday, July 30

SESSION CHAIRMAN: Chuck Lawson

8:30 - 9:30 am	**Kahan, W., "The Programming Environments' Contribution to Program Robustness"
9:30 - 10:15	Hull, T. E., "A Hardware Unit for Decimal Arithmetic with Controlled Precision"
10:15 - 10:30	BREAK
10:30 - 11:15	Ginsberg, Myron, "A Review of Performance Comparisons for Supercomputers and Conventional Machines" (The title in the proceedings is "A Brief Guide to the Literature on Supercomputers")
11:15 - 11:25	Delves, L. M., "The Use of Extensible Languages for Mathematical Software"
11:25 - 1:30 pm	BREAK

SESSION CHAIRMAN: Bob Mercer

1:30 - 2:00	**Hanson, David R., "A Portable File System"
2:00 - 3:00	**Osterweil, Leon, "TOOLPACK - A Collection of Tools for Mathematical Software"
3:00 - 3:15	BREAK
3:15 - 3:45	Mehlschau, Edward, "SOFTOOL 80™ A Methodology and Integrated Collection of Tools for Software Management, Development, and Maintenance"
3:45 - 4:15	Krogh, Fred T., and *Snyder, W. Van, "A Support Environment for Software Tools"
4:15 - 4:45	Boley, Daniel L.; *Gropp, William D.; and Theimer, Marvin M., "A Method for Constructing Preprocessors"
5:00 - 7:00	HAPPY HOUR
7:00 - ?	BANQUET

Friday, July 31

SESSION CHAIRMAN: Nelson Beebe

8:30 - 9:20	**Feldman, Stuart, "Mechanizing Operations on Source, Object, and Test Data - Why Should You Do All The Work?"
9:20 - 10:15	**Hennell, M. A.; Riddell, I. J.; and Woodward, M. R., "A Mutation Analysis of Numerical Software"
10:15 - 10:30	BREAK
10:30 - 10:50	Cline, A. K., "A Course on Mathematical Software"
10:50 - 11:10	Cioni, G.; Miola, A; and Ruffi, A., "Moving Software Systems to a Minicomputer"
11:10 - 11:30	Dodson, David S.; Lewis, John Gregg; and Poole, William G., Jr., "Tailoring Mathematical Software for the CRAY-1" (Presentation by Ivor Philips)
11:30 - 11:50	*Hanson, R. J. and Krogh, F. T., "Flexibility in Mathematical Software Development Using Option Arrays"

CONFERENCE OVERVIEW

FRED T. KROGH, Conference Chairman

The mathematical software community has as its primary goal the development of high quality tools for the solution of a wide variety of mathematical problems. The development of such tools is enhanced by other tools; both hardware and software, which provide a hospitable computing environment. Unfortunately, our environmental needs are not well known to the computing community at large. Under the able leadership of C.L. Lawson, JPL and ACM-SIGNUM have previously sponsored two conferences related to this one: "Workshop on Fortran Preprocessors for Numerical Software" in 1974, and "Conference on the Programming Environment for Development of Numerical Software" in 1978. These conferences have provided a unique forum for those interested in the environment for the development and use of mathematical software.

In the early days of computing there was an emphasis on the needs of those who were solving mathematical problems. But the perception that mathematical computation is only a small part of the marketplace and the fact that the needs of those engaged in numerical computation are not well known has led to considering those needs as little more than an afterthought. Recent advances in software and hardware technology are making it economical to create computing environments appropriate for specialized applications. The implementation of a small specialized language or operating system environment is now a small enough job that significant experimental environments can be designed and implemented. Large vector and array processors are being designed and built with numerical computation as the primary or only application. At the microprocessor level we are getting arithmetic units superior to what we are used to on general purpose computers. The computing environment for mathematical software is changing for the better. This conference provides a view of those changes.

Fortran still appears to be the language of choice for scientific and engineering computation. Interchange between the Fortran standards committee, X3J3, and the mathematical software community has proven valuable to both at previous conferences in this series. Continuing this tradition of interaction, talks by Adams, Meissner, Wilson, Wilkens, and Smith from X3J3 give an idea of the substantial changes that are being planned for Fortran. Some of these new features are being introduced primarily to meet needs in numerical computing.

There has been a lot of recent work on Fortran programming environments. Miller gives an overview of three of these and describes some current work. Lucas examines the environment from the viewpoint of one supporting the use of mathematical software in solving scientific and engineering problems. There is considerable overlap between statistical and mathematical software. But there has been considerably more emphasis on canned packages for the unsophisticated user in the former case. Ryan describes several statistical packages, where the emphasis has been on their packaging. He expects benefits from future interaction with computer science.

Amano, Chiba, Machida, and Maeda in one paper, and Gaffney in another, describe systems that make it easier for users of mathematical software to find the software appropriate for their needs.

At the 1978 conference, Kahan spoke on specifications for a proposed floating-point arithmetic standard, and Hull spoke on desirable characteristics for floating-point and elementary functions. Kahan's ideas have led to a proposed IEEE standard for floating-point. This proposed standard is currently available on one microprocessor chip, and there are strong indications that several others will be marketed soon. Expanding on the theme of floating-point arithmetic, Kahan discusses the programming environment's contribution to program robustness. Hull's ideas have also progressed to the point of being implemented. He describes the capabilities of his hardware unit here.

In these abstracts, Ginsberg gives a bibliography on supercomputers. In his talk he reviews performance comparisons for supercomputers and the more conventional type. Extensible languages have been found useful in designing the user interface for solving partial differential equations by Delves. He describes this work which was done in an ALGOL 68 environment.

In the 1978 conference, Webb Miller proposed the initiation of a collaborative effort to create a better programming environment for the Fortran programmer. His proposed "TOOLPACK" is now a project involving several institutions. Osterweil describes the TOOLPACK project. Mehlschau presents a commercial software tool collection available from SOFTOOL Corporation. Krogh and Snyder describe a support environment for software tools that is compatible with the requirements of the TOOLPACK project. A quick method for constructing preprocessors is described by Boley, Grupp, and Theimer.

Feldman asks, "Why should you do all the work?" and suggests mechanizing operations on source code, object code, and test data so that the programmer is relieved of many of the clerical tasks associated with programming. Hennell, Riddell, and Woodward describe a method for measuring the adequacy of test data. The method is based on testing whether errors deliberately introduced into the software (mutants) will be detected by the test data.

Cline's description of a course on mathematical software should be helpful to those teaching or planning such a course. Ciori, Miola, and Truffi present an aid for moving large programs to a minicomputer. And going in the opposite direction, Dodson, Lewis, and Poole consider the problem of tailoring mathematical software for the CRAY-1. Finally, Hanson and Krogh describe a way of passing optional data to a subprogram that has little impact on the user who has no need to pass such data.

Thanks are due: Webb Miller, Leon Osterweil and Brian Smith who helped in the selection of invited speakers; to Chuck Lawson who reminded me of a few oversights and helped with local publicity; to Kris Stewart for (wo)manning my phone while I was gone for three weeks prior to the conference; to Denise Chambers for secretarial support; and most of all to JPL for their generous support of this conference in terms of my time, secretarial support, and the printing of proceedings distributed at the conference.

FORTRAN STANDARDS, AN OVERVIEW

Jeanne Adams, NCAR
Chair, X3J3

FORTRAN 66

From the early beginnings in the fifties, FORTRAN was used as the de facto standard for scientific and engineering work. As its use spread, the potential for numerous imcompatible FORTRAN dialects became apparent. A rigorous definition of FORTRAN in a standard became necessary if continued use of the language were to be successful.

In 1960, The American Standards Association (ASA) established the committee for Computers and Information Processing. A subcommittee of this group was formed to consider common programming languages. In May of 1962, a working group began to study the possibility of introducing a standard for FORTRAN which was to become the first programming language to be standardized in the United States. Manufacturers and user groups sent representatives to the Standards Committee for FORTRAN (X3J3). They cooperated in the study and preparation of the first FORTRAN standard (X3.9-1966) which was released in 1966. Many areas were considered by X3J3 including current usage, clarity of syntax, ease of implementation and the language's potential for future extensions. Two clarifications were produced; one in 1967 and another in 1969.

FORTRAN 77

FORTRAN has always been a cost effective language; , what it may have lacked in elegance and style was gained in efficiency and ease of implementation. In 1967, the group responsible for maintenance of FORTRAN 66 concluded that it would require no more effort to begin on a revision than to continue producing official interpretations of the standard. In January 1969, X3J3 voted to begin work on a revision to X3.9-1966. The current standard X3.9-1978 took eleven years to complete; seven years were spent in preparing the draft, and four years in completing the approval process under the American National Standards Institute (ANSI) as it is now called. One significant result of this eleven year effort was that the FORTRAN 77 standard document is now more understandable than the 66 document. The standard itself contains six times more text in describing FORTRAN features than the 'old standard', resulting in improved readability and clarity. An important consideration in the development of FORTRAN 77 was the determination not to invalidate programs written in FORTRAN 66, but to ensure that these programs were upwardly compatible. A comparison chart of FORTRAN 66 and FORTRAN 77 will be summarized.

FORTRAN 8x

There is always resistance to change in programming languages and their associated standards. A programming language must reflect the needs of the user community and be responsive to the kinds of applications that ensure continuing popularity among users. At the time of final processing of FORTRAN 77, X3J3 had already begun the study of a design that would modernize FORTRAN. The objective was to keep pace with technology and state-of-the-art programming techniques and allow applications specific conventions to coexist and become part of a family of standards for FORTRAN.

In these early studies during 1978 and 1979, the committee studied language features from many perspectives. The emphasis was to examine the language as a whole, and how various candidate features were related. Surveys of various user groups were taken, and these needs were placed in a broad enough context so that condidate features for the standard could be chosen within a carefully considered framework. In certain areas (i.e. data structures and program form), development was necessary because of a need for a function not yet provided in the marketplace. An architecture is proposed (described in more detail in the next presentation on FORTRAN) that will place certain general features in the core, while others will be separated into modules. Interfacing to special purpose applications such as graphics or data base schemes is an important part of the technical proposal for the next revision. An important question concerns the ability to identify obsolete features and move these out of the language in a graceful way.

A basic array processing proposal has been completed and will be in the new revision. In the various surveys on user needs, the ability to manipulate arrays as fundamental objects was a very popular request.

A need has been expressed for a general precision data type facility to specify minimum precision for floating point operations. Numerical analysts have made important contributions to the precision proposals for the current revision, as well as to the Environmental Inquiry proposals which are discussed later in this session.

Liaison Activities

Liaison has been established with a number of national and international organizations. One of the very important liaison activities continues to be with SIGNUM. Nationally, X3J3 has established a liaison with the Graphics Standards Committee, the CODASYL Data Base Group, and the Purdue Workshop for Industrial Real Time FORTRAN. Internationally, we have a working rela-

tionship with the British Computer Society, the European Computer Manufacturer's Association, and Standards Groups from a number of countries such as Austria, Germany, Sweden and Canada.

MILESTONES

X3J3 has just completed the first draft of a document that contains all of the proposals passed so far for the next revision. The document is not intended to be text for the standard, but is is organized into sections such that the editors can prepare text easily. All new proposals before the committee are to be modifications of this document (Standing Document 6). In this way, members will have a single source of material for the next revision.

Many people ask what x is in 198x. However, it is difficult to answer this question. X3J3 has a milestone chart, but it is a working scheme for establishing committee objectives rather than any fixed promise for completion. Dates that occur before today have been modified to reflect actual progress, while dates that follow today reflect the committee's goals. A sample milestone chart has been prepared for this session.

CORE-AND-MODULES DESIGN FOR NEXT FORTRAN STANDARD

Loren P. Meissner
Lawrence Berkeley Laboratory

The Problem

Fortran isn't going to go away, no matter how much we deplore its irregularity and inelegance. Fortran 77 introduced the block-if and the zero-trip DO, and de-standardized Hollerith. But still nobody considers Fortran 77 a really modern programming language.

Can Fortran be modernized? If Fortran were modernized, would it still be Fortran? Would it have any advantage over Pascal? And what would happen to the billions of dollars of investment in "old Fortran" programs?

There is pressure from yet another direction to influence the future of Fortran. This is the need for major extensions to the language for the sake of a specialized application area (graphics, real-time, list processing, array processing, or CODASYL data base, for example). Incorporating any one of these major extensions erodes the practical usefulness of the language for anyone outside the specialized application area. To accommodate them all in a single language would obviously be impractical.

A Proposed Solution

The ANSI Fortran Standards Committee, X3J3, is by now firmly committed to the idea that a "core and modules" approach can solve this problem. The idea is to define a "Core Fortran" language, consisting of all of the "good" features of Fortran 77, plus modern replacements for the "obsolete" features of Fortran 77. The obsolete parts of Fortran 77 would, however, be retained in an "obsolete features module".

Other "application area support modules" would satisfy the need for major extensions needed for specific application areas.

Another kind of module, called a "language extension module", could add "fancier" features of broad application to the language (for example, a macro facility).

Structure of the language (see Figure 1). By the end of the life cycle of the next revision of the Fortran standard (in the mid to late 1990's), it is assumed that the "obsolete" features would no longer be heavily used so that the typical minimum-configuration Fortran compiler would support only Core Fortran, that is, the "good" features inherited from Fortran 77 plus the replacements for the "obsolete" parts of Fortran 77. However, early compilers for the new language would necessarily support all of Core Fortran plus the obsolete features module.

Extended compilers would also support the language extension module (or modules).

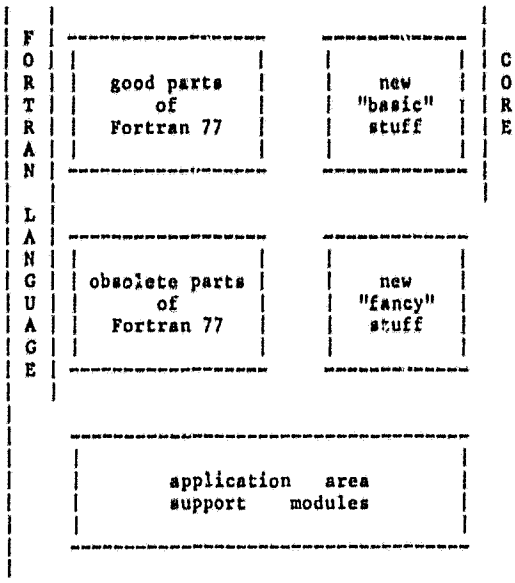
It is contemplated that the next Fortran standard will include both parts of Core Fortran, the obsolete features module, and perhaps one or more extension modules. It may be possible to add extension modules between revisions of the main Fortran standard. Application area support modules would be handled as separate "collateral" ANSI standards, and in many cases these would be developed by a more or less independent "task group" reporting to the X3J3 committee, rather than by the full committee.

Summary

Will the result be Fortran, or will it be a new language? It will certainly be a new language; the remarkable thing, if X3J3 is successful, will be the creation a new language that is compatible with its predecessor.

Will it be Fortran? Maybe; but only if X3J3 can maintain the self-discipline to keep Core Fortran a reasonably small language that is at least as easy to implement and to use as earlier versions have been.

FIGURE 1.
Fortran Core-and-Modules Structure



Parallel Processing in Fortran

Alan Wilson

ICL, London

The ANSI Fortran committee X3J3 has recently accepted proposals for parallel processing in Fortran. These proposals have been used to express algorithms such as Batcher sort, FFT, telephone network simulation, black/white SOR, image processing, noise reduction, least squares, solution of linear equations, assignment problems, matrix inversion,...

An attempt will be made to assess the completeness of the proposed language extensions:

> Do they seem to form a basis for the expression of (known) parallel algorithms?
>
> Are they likely to be useful in the search for new parallel algorithms?

GLOBAL DATA SHARING AND INTERNAL PROCEDURES IN FORTRAN 8X

Edward J. Wilkens
Perkin-Elmer
Tinton Falls, New Jersey

INTRODUCTION

In past actions, X3J3 has voted to remove certain features from CORE FORTRAN, including storage association COMMON,EQUIVALENCE,multiple entry points, extended DO loops and statement functions. It has also voted to add, or considered for addition, such features as GLOBAL, and internal subroutines.

This paper presents some proposed facilities to replace those removed.

The syntax described is consistent with proposals previously passed and currently proposed in April 1981. The area of data sharing and internal subroutines has been a rather fertile one for proposals, with several varied names and constructs proposed and discarded. These do share a significant number of common functions, if not name and syntax, and differ primarily because they spring from different sources of inspiration.

DATA SHARING

COMMON has been removed from the core of FORTRAN 8X primarily because of the storage association properties implied. However, there is no intention to remove the functionality of a fast, efficient global sharing mechanism provided by COMMON. The replacement for COMMON, called GLOBAL, eliminates storage association. It also provides for a single DEFINITION of the GLOBAL data, with multiple uses of that definition. The following is a typical definition:

```
GLOBAL DEFINITION /Shared_data/Real_var,Integer_var,Char-var
```

As the example implies, any mixture of types is allowed. No specific order of appearance or storage allocation is implied by the list of data elements.

In order for a program unit other than the one containing the GLOBAL DEFINITION to access the GLOBAL data, it must contain a GLOBAL statement naming the GLOBAL name (or blank) to be shared. For example:

```
GLOBAL /Shared_data/
```

provides a program unit other than the one containing the GLOBAL DEFINITION with access to the data elements in Shared_data by their name. No further definition of those variables is needed. They have all the attributes of their declarations in the defining program.

Dependent Compilation

All program units sharing a GLOBAL Data Area may be separately compiled. However, the program unit containing the GLOBAL DEFINITION must be compiled before the others. This places a burden on the processor to communicate in some processor dependent fashion the declared variables in the GLOBAL area, as well as their types, dimensionality, and locations in the GLOBAL area. It also places a burden on the programmer (or some processor defined user aids) to recompile any program unit containing GLOBAL after recompilation of a GLOBAL DEFINITION in which some information used by those dependent program units were changed.

FORTRAN 77 Compatibility

The primary difference between GLOBAL and COMMON is storage association and definition/use distinction. By providing a similar but differently named functionality, they may be freely mixed in the presence of the Obsolete Features Module. GLOBAL and COMMON may appear in the same program unit, provided they refer to different data areas.

INTERNAL PROCEDURES

Several features removed from CORE FORTRAN in past X3J3 actions include control constructs such as Statement Functions, ENTRY Statements and Alternate Returns. The facilities provided by these constructs remain to be provided in safer, more consistant constructs. Internal Procedures are provided for this purpose, with primary functionality including:

Replacement of the Statement Function
Replacement of extended range of DO LOOP
Remote code blocks
Replacement for ENTRY

An Internal Procedure is either an Internal Function or an Internal Subroutine, which are similar to the familiar Function and Subroutine that serve as program units in a FORTRAN program.

Internal Procedures are located at the end of a host program or procedure, immediately before the END statement or another Internal Procedure. Declarations for dummy arguments of an Internal Procedure are contained within it.

Internal Procedures are invoked from within the host procedure in an identical manner to external procedures.

Name Scoping

Goals for Name Scoping rules include regularity, convenience of sharing as in Statement Functions, safety from accidental error and replacement of error prone ENTRY argument rules. These goals tend to be somewhat at odds, and have resulted in several proposals and abandonments. An INHERIT statement with a list of names is provided to declare what entities are to be known in the Internal Procedure from its host. All other entities appearing in an Internal Procedure are local to it. An INHERIT ALL statement has been proposed, but not yet passed.

The INHERIT statement provides a regular, safe method for sharing data from the host. It does not provide the convenience of sharing as in Statement Functions. A replacement for a Statement Function would require an INTERNAL FUNCTION, INHERIT ALL assignment, and END INTERNAL statements.

Packages and Libraries

ENTRY statements have often been used in large libraries. Problems occur because of the difficulties of enforcing inaccessibility of dummy arguments from inactive ENTRYs. ENTRY INTERNAL Procedures have been proposed to declare an Internal Procedure callable from outside the Host. This allows a much safer functionality for ENTRY.

Large libraries are normally not compiled at one time. A Procedure may be declared a PACKAGE, which means that it is to be considered to be Host to Internal Procedures which may be separately compiled. Which Internal Procedures are to be so considered is specified in a CONTAINS Statement. Finally, the PACKAGE Procedure may be declared to be pure sharable data with no executable code by declaring it to be a SHELL Procedure.

Data Sharing Revisited

A SHELL Procedure may be seen to be equivalent in functionality to GLOBAL Data. Note that ENTRY, PACKAGE, and SHELL have not yet been approved, whereas GLOBAL Data has. Since ENTRY and PACKAGE are especially desirable features, especially for Mathematical Libraries, SHELL is an obvious extension to eliminate some almost redundant functionality (GLOBAL). However, GLOBAL's strong resemblance to COMMON makes it attractive. Final resolution of these issues remain to be decided.

A CONSUMER'S REPORT ON FORTRAN PROGRAMMING ENVIRONMENTS

Webb Miller
Department of Computer Science
University of Arizona
Tucson, Arizona 85721

In this talk we will summarize the attributes of several available Fortran programming environments, including the WATFIV compiler and run-time system, the Unix operating system and SOFTOOL 80. The strengths and weaknesses of each will be discussed. In addition, mention will be made of a few available free-standing software tools and of some planned developments.

WATFIV has long been regarded as an excellent system for debugging Fortran programs. This reputation is well deserved: the error messages are exceptionally clear and the automatic run-time checks are quite useful. The system supports a profiler and a preprocessor for a "structured" Fortran. Moreover, an interactive version of WATFIV that includes a symbolic debugger is available.

While WATFIV is not without notable omissions and undesirable features in the way it handles Fortran, its main deficiency is its limited scope and machine dependency. The user is left to the mercy of the native editor, file system, etc., and these are often relatively primitive.

Unix is a highly successful operating system developed at Bell Laboratories and now in place at over 2500 installations world-wide. It is particularly noteworthy as a program development environment for projects involving small numbers of programmers. Some of its more appealing attributes are a no-nonsense command language, a very clean file system with automatic updating of derived files, excellent text-processing capabilities including phototypsetting of mathematical equations, plus a host of software fragments and mechanisms for connecting the fragments to form organized tools.

Whereas earlier versions of Unix supported Fortran only marginally, recent versions (e.g., Berkeley System Distribution 4.0 for the PDP VAX-11/780) support it handsomely. Included are a Fortran preprocessor that, in my mind, stands above its many competitors, a symbolic debugger and a Fortran structurer.

Perhaps the main weakness of the Unix system as a whole, beside its machine dependency, is that it has the steep learning curve that you would expect of an operating system designed for expert programmers. The Fortran component of Unix currently suffers from a fair number of glitches because of its newness, a complaint that should be resolved with time.

SOFTOOL 80 is an integrated collection of software tools that aim to support a particular methodology for Fortran programming. It includes not only the expected components for static checking, preprocessing structure code, profiling and test coverage reports, but also tools to enforce certain programming standards and documentation formats set by management.

Further information about WATFIV, Unix and SOFTOOL 80 can be obtained from the following sources.

WAT NEWS
Computer Systems Group
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

Western Electric Co.
Patent Licensing Manager
P.O. Box 20046
Greensboro, N.C. 27420
(919) 697-6530

SOFTOOL CORPORATION
340 South Kellogg Avenue
Goleta, Ca. 93117
(805) 964-0560

Numerical Software - View from the Trenches

L. W. Lucas
Naval Weapons Center

This paper relates experiences of the author over the past seven years as Numerical Mathematics Coordinator at the Naval Weapons Center Central Computing Facility, evaluating, selecting, maintaining and marketing numerical software, and providing consulting, training, and documentation services. The Center is a consumer, not a producer, of numerical software. Applications include missile design and simulation, radar analysis, signal processing, detonation physics, and chemical kinetics.

Tentative Outline:

Implementing a numerical software library

- awareness - Mathematical Software I & II
- starting point - IMSL, JPL library
- literature survey
- additions - GEAR, DEPAC, EISPACK
- NESC test site - LINPACK, MINPACK

Marketing efforts

- documentation - Guide, Examples
- short courses - Matrix computation
  Numerical solution of ODE'S
  Computing random numbers
  Curve fitting
  Least squares
- academic - Numerical Methods

Suggestions to numerical software developers

- user interface
  - naming
  - documentation
  - reverse communication
- tradeoffs
  - change vs. stability
  - flexibility vs. ease of use
- constraints in engineering situations
  - sample data systems

PACKAGING STATISTICAL SOFTWARE

Thomas A. Ryan, Jr.

Most statistical computations are done using widely available statistical packages. In this paper, I will discuss primarily the SPSS, SAS, BMDP, P-STAT and Minitab systems, which together account for a high percentage of all statistical computing. Other packages which have influenced the development of statistical packages include Datatext, Genstat, IDA, IMPRESS, MIDAS, Omnitab, OSIRIS, SNAP, Statjob.

Development of statistical packages began with the BMD programs at UCLA, in the late 1950's. These programs formed an integrated set of procedures, but were not a statistical "package" in the current meaning of the word. They also had a very primitive user interface (e.g., to take the log of a variable, the user punched an "03" in card columns 10 and 11). In 1968, work began on a new series of BMD programs, called the P or parameter series, which identified input parameters with English keywords.

The initial design of the widely used SPSS system was begun in 1965. The publication and wide distribution of the SPSS manual by a commercial publisher in 1970 marks the beginning of statistical packages as we know them today. This manual put computing power in the hands of a very wide audience; the design of the SPSS system allowed even very computationally unsophisticated researchers to use (and abuse) the system.

Most statistical systems began in universities. Some (SPSS, SAS, P-STAT, IDA) eventually went commercial. Hardware manufacturers have had little impact.

Packaging

An important aspect of statistical software is a strong emphasis on the packaging. Statistical packages attempt to provide a total environment for data analysis, including data management, vector (and sometimes matrix) arithmetic, graphics, a collection of statistical procedures, and often report writers.

The mathematical algorithms tend to be relatively simple and a small part of the overall system. Computing time for typical procedures (such as least-squares regression) is often less than the computing time for inputting the data. The routines for output often occupy much more space than the key computation routines. As an example, LINPACK routines to do regression by the QR decomposition, plus routines to call them, total about 100 lines. Minitab's REGRESS command contains over 1,000 lines. The largest portions are for creating useful, readable printed output.

Portability

Most statistical packages are portable to some degree. (SAS, which runs only on large IBM computers, is a notable exception.) Most packages are undergoing active development, and issue a release with major enhancements (some of which permeate the entire system) every year or two. Each release must be implemented on 5, 10, or more brands of computers. Most go on mainframes such as IBM, CDC, UNIVAC, Burroughs, Honeywell and DEC, and supermini's such as VAX and PRIME. BMDP and Minitab also go on 16-bit minis such as PDP-11 and HP-3000.

The major difficulties encountered in porting these systems are: (a) their size, which is typically 100,000 lines of Fortran code and 1,000 subroutines, which pushes limits in linkage editors and requires heavy overlaying on some computers; and (b) the critical importance of the speed of input and the size of output routines. Neither of these problems are encountered in subroutine libraries.

The most primitive method of maintaining a (more or less) portable package is to develop the package on one computer and rely on conversion centers to adapt the program to other computers. Feedback from converters is used to improve portability. This method is still used by SPSS. A simple yet powerful way to handle portability problems was developed by Roald Buhler for the P-STAT system in 1971. Under his approach, a master source is maintained with all versions present. A simple selection preprocessor chooses the appropriate version based on codes in the first few spaces of the lines. This approach has also been successfully adopted by Minitab and a slightly modified version has been adopted by BMDP and IMSL.

Interaction with Computer Science

The developers of statistical packages are generally not computer scientists, but have learned most of the computer science they know "on the job". The principal developers of some packages (e.g., BMDP, SAS, Minitab) are statisticians, for others (e.g., SPSS) the developers are consumers of statistics (social scientists for SPSS). Some of the developers are involved in statistical consulting (e.g., BMDP, Minitab), and all packages get extensive feedback from users. This has tended to encourage sensible, realistic refinements and extensions to the packages -- but probably has focused effort near the existing capabilities of the package rather than on developments in totally different directions.

Computer scientists have obviously influenced the designs of the packages and the algorithms included in them. Equally obvious to users is that the influence has not been as great as it should have been. For example, there are still numerically unstable algorithms in packages, and even more inefficient, brute force methods used to prevent instability. One reason for the lack of influence has been that statistical package developers have often been forced to face problems (e.g., in portability and user interface) before computer scientists were prepared to give answers.

Statistics package developers should make much more use of what computer scientists know about development of reliable software, about user interface, about discrete algorithms, and so on. This process would be aided if more computer scientists studied some of the interesting problems faced by package developers. The most important forum for exchange of ideas is the series of meetings, Computer Science and Statistics: Annual Symposium on the Interface.

Algorithm Bank: Information System for Mathematical Software

Kaname Amano, Maaki Chiba, Akeno Machida * and Takashi Maeda **

* Computing Center
** Department of Engineering Science
Hokkaido University, Sapporo 060, Japan

1. Introduction

A large number of algorithms have been proposed for scientific computations and some of them have been improved subsequently. These are published in many publications, and it is not easy for general users to find suitable algorithms for their purpose. We intend to construct an information system for mathematical software of fine quality. Rice (1) pointed out the strengths and weaknesses of two approaches for user interface with algorithms: one in the program libraries and the other in the extended systems available as an integral part of the programming languages. We present here a basic idea of an approach of the former type based on an information retrieval method combined with some artificial intelligence techniques. A preliminary implementation is also reported. It will support the following steps in designing and programming of problem solving by computers:

- setting up the problem in an appropriate form
- designing the algorithm
- programming
- debugging and execution of the program
- analysis of the results

The retrieval system should deal with the following two kinds of information:

- information concerning individual algorithm
- information concerning a set of algorithms for a specific purpose. This kind of information will be useful for selecting a suitable one, and is called algorithmic knowledge in this paper.

2. Information Representation of Algorithms

Algorithms may exist in various forms, i.e., load modules, source programs in widespread or rather uncommon programming languages, natural language description with mathematical notation, etc. One may need various kinds of information for identifying algorithms, e.g., bibliographic items, specification of functions, usage of programs, etc. One of the two alternatives must be chosen to describe the function: the first, procedural description by higher level languages or conceptual description by technical terms; the second, description with several formats each for a specific problem field or description with a single format for all problem fields. We adopt the conceptual description with a single format.

The information representation of individual algorithm consists of the following three types of attribute-value set:

(1) bibliographic attributes, i.e., the journal, volume, number, page, year, title, authors, classification codes, keywords and keyphrases, etc.
(2) functional attributes, i.e., the problem domain, method, performance, etc. The performance is evaluated by CPU-time, number of operations, memory required, accuracy, robustness, etc.
(3) operational attributes, i.e., the programming language, usage of programs, tested or not tested, etc.

The values for these attributes are mostly the technical terms in mathematics and computer sciences. Syntactic rules are needed for description such as treatment of special symbols, tags, descriptors, standard forms, etc.

This information representation scheme can be regarded as a kind of indexing method for algorithms. Problem-oriented retrieval becomes possible to some extent even with usual information retrieval methods. But a mixed-initiative dialogue system will be more useful for common users. Then the system should deal with the following types of information as the algorithmic knowledge:

(1) knowledge on terms, i.e., the relation of synonym, superclass and subclass of terms, etc. Short comments are also useful to assist the users.
(2) higher level knowledge
  - mathematical knowledge, for example matrix inversion can be regarded as equivalent to solving simultaneous linear equations, etc.
  - empirical knowledge, for example matrix inversion may be avoided whenever possible, etc.

It is possible to assist users by prompting at the selective points with the algorithmic knowledge. This can be implemented by the usual information retrieval method combined with some artificial intelligence techniques, say the production system, etc. We can define the higher level knowledge as a mapping between representative terms of functional attributes. Rice (2) discussed the algorithm selection problem and proposed a mathematical model with the problem space (in some cases feature space and criteria space are also considered), algorithm space and performance measure space. These spaces may correspond to the above functional attributes. Our information retrieval method is less mathematical and more empirical.

3. A Bibliographic Database

As the first step of the development of our system, we have constructed a bibliographic database of CALGO (Collected ALGOrithms from acm) which includes certifications and remarks for some algorithms. In CALGO, there are three periods characterized by the start of collection, the claim for quality of each algorithm and efforts for accessibility to the collection. Some of the proposed algorithms are improved subsequently by remarks. The publication of CALGO in the loose-leaf form is an approach to the accessibility problem. An information retrieval approach by computer will be more efficient to this problem. In our system, the certifications and remarks are merged into the original algorithm to supply the augmented information. This database is now available at the Hokkaido University Computing Center with the retrieval system ORION (Online Retriever of InformatION which is available on a HITAC computer). Fig.1 shows an example of the retrieval process.

```
                KEYWORDS     NO PREFIX 3001810624
ENTER YOUR REQUEST
   1/ FIND SHIFT
*       14    1/  SHIFT
   2/ FIND LINEAR AND EQUATION*
*       36    2/  LINEAR
*       47    3/  EQUATION (  2   TERMS COMBINED)
*       15    4/  LINEAR AND EQUATION*
   5/ FIND OVERDETERMINED
*        4    5/  OVERDETERMINED
   6/ FIND 1 AND 5
*        3    6/  1 AND 5
   7/ DISPLAY

ITEM 1

ALGORITHM 328
 REMARKS     1
 TITLE    CHEBYSHEV SOLUTION TO AN OVERDETERMINEI
 YEAR1    8JUNE1967 AND 22NOV,1967
 AUTHOR1  RICHARD H,BARTELS AND GENE H,GOLU  (COI
          UNIVERSITY,STANFORD,CALIF,94305)*
```

Fig.1 An example of retrieval of the database CALGO

Some source programs, though not tested yet, are stored on disks and can be used online. We intend to construct a mixed-initiative dialogue system for some specific problem fields as the second step of the development.

4. Concluding Remarks

We have presented a basic idea of the information system for mathematical software. A large number of quality algorithms are now available but are not utilized for many users. The information retrieval approach will be effective for dissemination problem of quality mathematical software.

references

(1) J.R.Rice: The Distribution and Sources of Mathematical Software, in Mathematical Software, edited by J.R.Rice, Academic Press, 1971.

(2) J.R.Rice: The Algorithm Selection Problem, in Advances in Computers, vol.15, edited by M.Rubinoff et al., Academic Press, 1976.

SOFTWARE MANAGEMENT

P. W. Gaffney

Computer Sciences Division at*
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37830

The purpose of this talk is to describe two software aids, called NASTI and NIT,[1] that have been developed at Oak Ridge for managing numerical software. As part of our comprehensive numerical software service, we have acquired a large collection of quality routines from sources other than mathematical libraries. In order to avoid duplicating the acquisition of a piece of software and at the same time provide users with information about existing software, we have developed an interactive data base called NASTI. This data base is managed by the System 1022 Data Base Management System[2] and is available on our PDP-10 computer. The information contained in NASTI has been deliberately kept to a minimum. Thus, for each piece of software described in NASTI, a computer user has access to the following quantities:

NAME	- Name of the piece of software
PROBLEM	- The problem area that the software is suitable for, e.g., Partial Differential Equations
PURPOSE	- A brief description of what the software purports to do and some advice on the recommended use
METHOD	- The main numerical methods used
ORIGIN	- The source of the software
VERSION	- A date which usually signifies when the software was acquired
LOCATION	- The location of the FORTRAN source of the software on our system

These quantities may be regarded as keys which the user may employ during a particular search sequence. A disadvantage of NASTI is that the language of the 1022 system is clumsy for the casual user. A further disadvantage is that since the 1022 system is not widely available, NASTI is non-portable. However, NASTI is an example of an aid which was constructed using existing facilities, and which adequately assists in the management and dissemination of information about numerical software.

In order to provide users with advice on the correct choice of a numerical routine for a particular problem, we have developed a numerical interactive tree called NIT. During a NIT session, a computer user is asked certain questions in an attempt to identify the particular routine or group of routines which are best suited for solving the user's problem. By responding to these questions, the user is led effortlessly to a recommendation. At present, NIT forms the basis of a system which is being developed for on-line documentation. Thus, NIT has the capability of providing HELP files and also gives sufficient information to enable a user to incorporate the recommended software in a FORTRAN program. Moreover, NIT also gives information on how to execute this program on the various computers available at Oak Ridge. Unlike NASTI, NIT is portable, because it has been written to conform to the PFORT[3] verifier. Thus, NIT may be installed on a variety of computers.

The talk will contain a brief description of NASTI and an explanation of the development of NIT together with proposed extensions.

---

[1] S. W. Gaffney, J. W. Wooten, and K. A. Kessel, "NIT - A Numerical Interactive Tree," ORNL/CSD/TM-139.

[2] "System 1022 Data Base Management System," Software House, Cambridge, MA 02138.

[3] "The PFORT Verifier," Software Practice and Experience, 4 (1974), pp. 359-377.

*Operated by Union Carbide Corporation under contract W-7405-eng-26 with the U.S. Department of Energy.

The Programming Environment's Contribution to Program Robustness

W. Kahan
University of California
Berkeley

A robust program to solve a quadratic $ax^2 - 2bx + c = 0$ will conceal from its user any over/underflow in the discriminant $b^2 - ac$ while revealing over/underflow just when a calculated root lies out of range. In general, robust programs conceal spurious exceptions from their users while rendering faithfully those exceptions pertinent to final results. If this assertion characterizes program robustness truly, then robustness is impractical in most programming environments and a challenging task even in so favourable an environment as is specified by the proposed IEEE standard for floating point arithmetic.

A Hardware Unit for Decimal Arithmetic with Controlled Precision

T.E. Hull, Department of Computer Science
University of Toronto

Introduction

The main purpose of this paper is to describe briefly the capabilities of an arithmetic unit called CADAC (for Clean Arithmetic with Decimal Base and Controlled Precision) which is currently being constructed [1] at the University of Toronto.

The unit is intended to support language facilities (including exception handling, and programmer control of the precision and exponent range of the operands, as well as of the operations performed on the operands) such as have been advocated by the author [see, e.g., 2]. Previous attempts to implement these ideas have been based on preprocessors, which suffer from shortcomings in terms of both flexibility and efficiency. The building of CADAC is intended to provide hard data on what trade-offs are involved if the basic ideas are supported by the hardware.

It is intended that CADAC be interfaced initially with a PDP-11/34, whose 16-bit wordlength has therefore influenced the design.

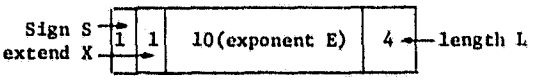
The language facilities

As explained in more detail elsewhere (see [2] and the references given there), the proposed language facilities allow the programmer to specify separately the precisions of the operands (along with the exponent ranges), and the precision (and exponent range) of the operations to be performed on the operands. There are two main ways in which the programmer can take advantage of this capability: (1) one is to be able to carry out intermediate stages of a calculation with a precision and/or exponent range that is higher than the operands; (2) the other is to be able to repeat a portion of a calculation with higher and higher precision and/or exponent range until some criterion (such as an error requirement) is satisfied. The second capability is the more complicated; the program outline given below (which illustrates the main features of an algorithm for solving equations to within a prescribed tolerance "tol") illustrates what we have in mind.

```
float(8) root   -- precision 8, default exp.
-----           -- other declarations, etc.
p = 8           -- initialize precision
flag = true
while (flag = true & p ≤ 32)
    begin precision(p)
        float(p) approximation
        -----
        find approximate solution
        find error bound
        if bound ≤ tol
            root = approximation
            flag = false
        end if
    end begin
    p = p+4
end while
```

Number representation, arithmetic, exceptions

The first 16-bit word of a memory location is interpreted by CADAC as shown in the diagram shown below:



The number of decimal digits in the normalized significand is 2L+2, and they are to be found in succeeding words, 4 per word. E is the excess-512 exponent. E = 0 is reserved for the value 0, unless S is negative, in which case the first 4 bits of the next word are hexadecimal F, E, etc., to represent "indeterminate", "not-yet-assigned", etc. If X = 1 the format becomes "extended" and both exponent and length, as well as the digits, are to be found in subsequent words.

Properly rounded floating-point arithmetic is carried out. It turned out to be relatively efficient to work only with multiples of 2 decimal digits. Other rounding modes are also provided, in particular so that interval arithmetic can be supported easily.

CADAC has a single accumulator, communicates with its host in DMA mode, and maintains an exception status register for the host. A 3-stage pipeline running at 10 Mhz handles 2-digit by 2-digit pairs, and multiplies two 32-digit numbers in about 30 microseconds.

Exceptions (including overflow, underflow, and roundoff) are flagged. Wraparound results are left after overflows and underflows, indeterminate after zero-divide.

Cost

Two large boards are used, each with close to 100 integrated circuits, including a mixture of SSI up to LSI, costing a total of about $4000 (Can.) for the two boards. Another few hundred dollars are needed for the interface board, power supply, cabinet and cables. It is expected that the total time required for the design, and for construction of the prototype, will be about 2 man-years (Can.).

References

[1] M. Cohen, V.C. Hamacher and T.E. Hull. CADAC: An Arithmetic Unit for Clean Decimal Arithmetic and Controlled Precision. IEEE Fifth Symposium on Computer Arithmetic, Ann Arbor, Michigan, May, 1981.

[2] T.E. Hull. Desirable Floating-Point Arithmetic and Elementary Functions for Numerical Computation. Proceedings Conference on the Programming Environment for Development of Numerical Software, 96-99 (SIGNUM Newsletter 14, edited by C.L. Lawson, 1979).

A Brief Guide to the Literature on Supercomputers

Myron Ginsberg
Computer Science Department
General Motors Research Laboratories
Warren, Michigan 48090

As 2-D and 3-D mathematical models come closer to reflecting real-world behavior, there is a substantial increase in the number of floating-point operations which must be performed as the grid structure is refined; for example, doubling the number of grid points in a 2-D problem produces a 4-fold increase in computing and for a 3-D problem in time there is a 16-fold increase. Machines in the emerging class of supercomputers offer some alternatives for parallel computation in attempting to deal effectively with such problems.

The bibliography given below provides a sampling of references to the literature associated with parallel algorithms (Section I), vector architecture (Section II), performance testing (Section VI), and specific supercomputers such as Cray Research's CRAY-1, (Section III), Control Data's Cyber 200 Series (Section IV), and the recently cancelled Burroughs Scientific Processor (Section V). The author welcomes readers to submit additional recent references in any of the aforementioned areas.

The oral presentation will focus attention on point-by-point comparisons of the CRAY-1 and Cyber 205. Emphasis will be placed on those attributes which directly affect the design and implementation of mathematical software for such supercomputers. Also, results will be presented from a variety of performance studies involving the CRAY-1, IBM 3033, and several other computer systems.

I. Parallel Algorithms

1. Brent, R. P., "The Parallel Evaluation of General Arithmetic Expressions," J. Assoc. Comput. Mach., Vol. 21, No. 2, 1974, pp. 201-206.

2. Chen, S. C., D. J. Kuck, and A. H. Sameh, "Practical Parallel Band Triangular System Solvers," ACM Trans. Math. Software, Vol. 4, No. 3, September 1978, pp. 270-277.

3. Gajski, D. D., "Solving Banded Triangular Systems on Pipelined Machines," Proceedings 1979 International Conference on Parallel Processing, August 1979, pp. 308-319.

4. Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, Vol. 20, No. 4, October 1978, pp. 740-777.

5. Kung, H. T., "The Structure of Parallel Algorithms," Advances in Computers, edited by M. C. Yovits, Vol. 19, Academic Press, New York, 1980, p. 65-112.

6. Miranker, W. L. "Parallel Methods for Solving Equations," Parallel Computers - Parallel Mathematics - Proceedings of the IMACS(AICA)-GI Symposium, edited by M. Feilmeier, North-Holland Publishing Company, Amsterdam, 1977, pp. 9-15.

7. Miranker, W. "A Survey of Parallelism in Numerical Analysis," SIAM Review, Vol. 13, 1971, pp. 524-547.

8. Ortega, J. M. and R. G. Voigt, Solution of Partial Differential Equations on Vector Computers, Report No. 77-7, ICASE, NASA Langley Research Center, Hampton, Virginia, March 30, 1977; also in Proceedings of the 1977 Army Numerical Analysis and Computers Conference, U.S. Army Research Office, Research Triangle Park, North Carolina, March 1977, pp. 475-525.

9. Poole, W. G. and R. G. Voigt, "Numerical Algorithms for Parallel and Vector Computers: An Annotated Bibliography," ACM Comput. Rev., Vol. 15, No. 10, October 1974, pp. 379-388.

10. Sameh, A. H., "Numerical Parallel Algorithms - A Survey," High Speed Computer and Algorithm Organization, edited by D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Academic Press, New York 1977, pp. 207-228.

11. Sameh, A. H. and D. J. Kuck, "Parallel Direct Linear System Solvers - A Survey," Parallel Computers - Parallel Mathematics - Proceedings of the IMACS (AICA)-GI, Symposium, edited by M. Feilmeier, North - Holland Publishing Company, Amsterdam, 1977, pp 25-30.

12. Stone, H. S., "Parallel Tridiagonal Equation Solvers," ACM Trans. Math. Software, Vol. 1, No. 4, December 1975, pp. 289-307.

II. Computer Architecture Considerations

1. Chen, T. C., "Overlap and Pipeline Processing," Introduction to Computer Architecture, edited by H. S. Stone, Science Research Associates, Chicago, Illinois, 1975, pp. 375-431.

2. Hallin, T. G. and M. J. Flynn, "Pipelining of Arithmetic Functions," IEEE Trans. Comput., Vol. C-21, 1972, pp. 880-886.

3. Kozdrowicki, E. W. and D. J. Theis, "Second Generation of Vector Super Computers," IEEE Computer, Vol. 13, No. 11, November 1980, pp. 71-83.

4. Kuck, D. J., "A Survey of Parallel Machine Organization and Programming," ACM Comput. Survey, Vol. 9, No. 1, 1977, pp. 29-59.

5. Kuck, D. J., D. H. Lawrie, and A. H. Sameh (eds.), High Speed Computer and Algorithm Organization, Academic Press, New York, 1977.

6. Lawrie, D. H., "Access and Alignment of Data in an Array Processor," IEEE Trans. Comput., Vol. C-25, No. 12, 1975, pp. 1145-1155.

7. Ramamoorthy, C. V. and H. F. Li, "Pipeline Architecture," ACM Comput. Survey, Vol. 9, No. 1, 1977, pp. 61-102.

8. Rudsinski, L. and J. Worlton, The Impact of Scalar Performance on Vector and Parallel Processors, Report LA-UR-76-2656, Los Alamos Scientific Laboratory, Los Alamos, New Mexico, 1976; summary in High Speed Computer and Algorithm Organization, edited by D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Academic Press, New York, 1977, pp. 451-452.

9. Sugarman, R., "'Superpower' Computers," IEEE Spectrum, Vol. 17, No. 4, April 1980, PP. 28-34.

10. Voigt, R. G., The Influence of Vector Computer Architecture on Numerical Algorithms, Report No. 77-8, ICASE, NASA Langley Research Center, Hampton, Virgina, March 31, 1977; also in High Speed Computer and Algorithm Organization, edited by D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Academic Press, New York, 1977, pp. 229-244.

III. Cray Research's CRAY-1

1. Ames, W. G., P. G. Buning, D. A. Calahan, D. A. Orbits, and E. J. Sesek, Sparse Matrix and Other High-Performance Algorithms for the CRAY-1, SEL Report No. 124, Department of Electrical and Computer Engineering, Systems Engineering Laboratory, University of Michigan, Ann Arbor, Michigan, January 25, 1979.

2. Asprey, M. W., "Vectorization from a Large Code Point of View," Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors, compiled by B. L. Buzbee and J. F. Morrison, Conference Proceedings LA-7491-C, Los Alamos Scientific Laboratory, Los Alamos, New Mexico, October 1978, pp. 16-40.

3. Buzbee, B. L., Implementation of Algorithms on the CRAY-1, report, Los Alamos Scientific Laboratory, Los Alamos, New Mexico August 1978.

4. Buzbee, B. L., G. H. Golub, and J. A. Howell, "Vectorization for the CRAY-1 of Some Methods for Solving Elliptic Difference Equations," High Speed Computer and Algorithm Organization, edited by D. J. Kuck, D. H. Lawrie, A. H. Sameh, Academic Press, New York, 1977, pp. 255-271.

5. Calahan, D. A., "A Block-Oriented Sparse Equation Solver for the CRAY-1," Proc. 1979 International Conference on Parallel Processing, August 1979, pp.116-123.

6. Calahan, D. A., W. G. Ames, and E. J. Sesek, A Collection of Equation-Solving Codes for the CRAY-1, SEL Report No. 133, Department of Electrical and Computer Engineering, Systems Engineering Laboratory, University of Michigan, Ann Arbor, Michigan, August 1, 1979.

7. Cray Research, CRAY-1 Fortran (CFT) Reference Manual, Publication No. 2240009 (Revision G) Cray Research, Inc., Mendota Heights, Minnesota, May 1980.

8. Cray Research, CRAY-1 Hardware Reference Manual, Publication No. 2240004 (Revision E) Cray Research, Inc., Mendota Heights, Minnesota, May 1980.

9. Cray Research, The CRAY-1S Series of Computers, Publication No. 2240008D, Cray Research, Inc., Mendota Heights, Minnesota, 1980.

10. Cray Research, Scientific Applications Package Handbook, (Revision B), Cray Research, Inc., Mendota Heights, Minnesota, January 1981.

11. Higbie, L., "Applications of Vector Processing," Computer Design, Vol. 17, No. 4, April 1978, pp. 139-145.

12. Higbie, L., Vectorization and Conversion of Fortran Programs for the CRAY-1 (CFT) Compiler, Publication No. 2240207, Cray Research, Inc., Mendota Heights, Minnesota, June 1979.

13. Johnson, P. M., "An Introduction to Vector Processing," Computer Design, Vol. 17, No. 2, February 1978, pp. 89-97.

14. Orbits, D. A. and D. A. Calahan, Data Flow Considerations in Implementing a Full Matrix Solver with Backing Store on the CRAY-1, Report No. 98, Systems Engineering Laboratory, University of Michigan, Ann Arbor, Michigan, 1976.

15. Petersen, W. P., "CRAY-1 Basic Linear Algebra Subprograms for CFT Usage," Technical Note No. 2240208 Cray Research, Inc. Minneapolis, Minnesota, February 1979.

16. Russell, R. M., "The CRAY-1 Computer System," Communications of the ACM, Vol. 21, No. 1, January 1978, pp. 63-72.

17. Sites, R. L., "An Analysis of the CRAY-1 Computer," ACM SIGARCH Newsletter, Vol. 6, No. 7, April 1978, pp. 101-106.

IV. Control Data's Cyber 200 Series

1. Control Data Corporation, CDC Cyber 200 Fortran Language 1.5 Reference Manual for Use with CDC Cyber 200 Operating System 1.5, Revision B, Publications and Graphics Division, Control Data Corporation, Sunnyvale, California, August 1980.

2. Control Data Corporation, CDC Cyber 200/Model 205 Computer System, Publicaion No. 60256020 (Revision 1), Control Data Corporation, St. Paul, Minnesota, September 29, 1980.

3. Control Data Corporation, CDC Cyber 200/Model 205 Technical Description, Control Data Corporation, Minneapolis, Minnesota, November 1980.

4. Hoffman, D. E., A Parameter Study of a Vectorized Chebyshev Algorithm on the CDC Cyber 203. Control Data Corporation, Pelham, New York, April 1980.

5. Kascic, M. J., Jr., "A Direct Poisson Solver on STAR," Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors, compiled by B. L. Buzbee and J. F. Morrison, Conference Proceedings LA-7491-C, Los Alamos Scientific Laboratory, Los Alamos, New Mexico, October 1978, pp. 137-165.

6. Kascic, M. J., Jr., Vector Processing on the Cyber 200, Control Data Corporation, St. Paul Minnesota, 1979; also published in Infotech State of the Art Report "Supercomputers", November 1979 and in Angewandte Informatik, January 1980, pp. 27-37.

7. Kascic, M. J., Jr., Vector Processsing: Problem or Opportunity?, Control Data Corporation, St. Paul, Minnesota, 1979; also published in IEEE COMPCON '80, November 1979.

8. Lambiotte, J. J., Jr., Effect of Virtual Memory on Efficient Solution of Two Model Problems, Technical Memorandum TM X-3512, NASA Langley Research Center, Hampton, Virginia, 1977.

9. Lambiotte, J. J., Jr. and R. G. Voigt, "The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computer," ACM Trans. Math. Software, Vol. 1, No. 4, December 1975, pp. 308-329.

10. Lincoln, N. R., "It's Really Not as Much Fun Building a Supercomputer as It Is Simply Inventing One," High

Speed Computer and Algorithm Organization, edited by D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Academic Press, New York, 1977, pp. 3-11.

11. Lincoln, N. R., A Safari through the Control Data STAR-100 with Gun and Camera, paper, Control Data Corporation, Arden Hills, Minnesota, 1978.

12. Mossberg, B., An Informal Approach to Number Crunching on the Cyber 203/205, Control Data Corporation, Cyber 200 Support, HSM02N, 1801 West County Road B, Roseville, Minnesota 55113, March 1981.

13. Noor, A. K. and R. E. Fulton, "Impact of CDC STAR-100 Computer on Finite Elements Systems," J. Struct. Div., ASCE, Vol. 101, 1975, pp. 731-750.

14. Noor, A. K. and S. J. Hartley, "Evaluation of Element Stiffness Matrices on CDC STAR-100 Computer," J. Comput. Structures, Vol. 9, No. 2, 1978, pp. 151-161.

15. Noor, A. K. and J. J. Lambiotte, Jr., "Finite Element Dynamic Analysis on CDC STAR-100 Computer," Computers and Structures, Vol. 10, No. 1, 1979, pp. 7-19.

16. Rothmund, H. J. and K. L. Murphy, Programming Methodogy for CDC Cyber 205 Vector Processor, Control Data Corporation, St. Paul, Minnesota, August 1980.

17. Redhed, D. D., A. W. Chen, and S. G. Hotovy, "New Approach to the 3-D Transonic Flow Analysis Using the STAR-100 Computer," AIAA Journal, Vol. 17, No. 1, January 1979, pp. 98-99.

V. Burroughs Scientific Processor

1. Burroughs Corporation, Control Program - Burroughs Scientific Processor, Burroughs Corporation, Paoli, Pennsylvania, November 1977.

2. Burroughs Corporation, Overview, Perspective, Architecture - Burroughs Scientific Processor, Burroughs Corporation, Paoli, Pennsylvania, February 1978.

3. Burroughs Corporation, Floating Point Arithmetic - Burroughs Scientific Processor, Burroughs Corporation, Paoli, Pennsylvania, December 1978.

4. Burroughs Corporation, Implementation of Fortran - Burroughs Scientific Processor, Burroughs Corporation, Paoli, Pennsylvania, November 1977.

5. Jensen, C. "Taking Another Approach to Supercomputing," Datamation, Vol. 24, No. 2, February 1978, pp. 159-172.

6. Stokes, T. A., "Burroughs Scientific Processor," High Speed Computer and Algorithm Organization, edited by D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Academic Press, New York, 1977, pp. 85-89.

VI. Some Performance Testing and Benchmark Results

1. Boley, D. L., "Vectorization of Block Relaxation Techniques - Some Numerical Experiments," Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors, compiled by B. L. Buzbee and J. F. Morrison, Conference Proceedings LA-7491-C, Los Alamos Scientific Laboratory, Los Alamos, New Mexico, October 1978, pp. 166-173.

2. Bucy, R. S. and K. D. Senne, "Nonlinear Filtering Algorithms for Parallel and Pipeline Machines," Parallel Computers - Parallel Mathematics - Proceedings of the IMACS (AICA) - GI, Symposium, edited by M. Feilmeier, North-Holland Publishing Company, Amsterdam, 1977, pp. 93-97.

3. Calahan, D. A., W. N. Joy, and D. A. Orbits, Preliminary Report on Results of Matrix Benchmarks on Vector Processors, Report No. 94, Systems Engineering Laboratory, Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, Michigan, May 1976.

4. Dongarra, J., Some LINPACK Timings on the CRAY-1, Report No. LA-7389-MS, Los Alamos Scientific Laboratory, Los Alamos, New Mexico, June 1978.

5. Fong, K. and T. L. Jordon, "Some Linear Algebraic Algorithms and Their Performance on CRAY-1," High Speed Computer and Algorithm Organization, edited by D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Academic Press, New York, 1977, pp. 313-316; also Report LA-6774, Los Alamos Scientific Laboratory, Los Alamos, New Mexico, June 1977.

6. Fornberg, B. "A Vector Implementation of the Fast Fourier Transform Algorithm," Math. Comp., Vol. 36, No. 153, January 1981, pp. 189-191.

7. Gentzsch, W., C. Weiland, and D. Müller - Wichards, Possibilities and Problems with the Application of

Vector Computers as Shown by the Numerical Processing of Several Physical Flow Problems, Report, German Research and Testing Establishment for Aerospace (DFVLR), April 2, 1980.

8. Hayes, A. H. and I. Y. Bucher, Los Alamos Scientific Laboratory Computer Benchmark Performance 1979, Report LA-8689-MS, Los Alamos Scientific Laboratory, Los Alamos, New Mexico, February 1981.

9. Hertweck, F., W. Schneider, and U. Schwenn, Benchmark Tests with the CRAY-1, IPP Report R/31, Max Planck Institute for Plasma Physics, May 1979.

10. Jefferson, T. H. and M. R. Scott, Megaflop Comparisons of Various Computers, Report SAND80-2205, Sandia National Laboratories, Livermore, California, October 1980.

11. Killough, J. E., "The Use of Vector Processors in Reservoir Simulation," Paper SPE 7673, presented at SPE-AIME Fifth Symposium on Reservoir Simulation, Denver, February 1979.

12. Mrosovsky, I., J. Y. Wong, and H. W. Lampe, "The Construction of a Large Field Simulator on a Vector Computer," paper SPE 8330, presented at the SPE-AIME 54th Annual Fall Technical Conference and Exhibition, Las Vegas, September 1979.

13. Nolen, J. S., D. W. Kuba, and M. J. Kascic, Jr., "Application of Vector Processors to the Solution of Finite Difference Equations," Paper SPE7675 presented at the SPE-AIME Fifth Symposium on Reservoir Simulation, Denver, February 1979.

14. Rothmund, H., Implicit Navier-Stokes Code on the Cyber 203/205 Computer, Control Data Corporation, June 6, 1980.

15. Rudsinski, L. and G. W. Pieper, Evaluating Computer Program Performance on the CRAY-1, Report ANL-79-9, Applied Mathematics Division, Argonne National Laboratory, Argonne, Illinois, January 1979.

16. Shang, J. S., P. G. Buning, W. L. Hankey, and M. C. Wirth, "Performance of a Vectorized Three-Dimensional Navier-Stokes Code on the CRAY-1 Computer," AIAA Journal, Vol. 18, No. 9, September 1980, pp. 1073-1079.

The Use of Extensible Languages for Mathematical Software

A Case Study

L. M. Delves

Department of Computational & Statistical Science
University of Liverpool, Liverpool, England

1. What and Why

GEM 2 is a general program for the solution of elliptic partial differential equations

$$-\nabla A(x)\nabla f(x) + B(x)f(x) + C^t(x).\nabla f = g(x) \qquad (1)$$

subject to a variety of boundary conditions, over arbitrary bounded or infinite two dimensional regions. It was written in the extensible language Algol 68, and this paper outlines the advantages of this approach. These depend strongly on the ability within this language to define not only new modes, but operators on these modes; and to include absolutely anything inside a structure, and to deliver absolutely anything as the result of a procedure. The advantages which accrue include:

a) Ease of writing. The program is written in terms of objects: lines, elements, differential equations, boundary conditions; which are natural to the mathematical problem.

b) Ease of debugging. These same features make it easy (well, easier) to locate and remove bugs.

c) Provision of a natural user interface is also simplified; it is possible to let the user see the highest level structures directly, and to provide a convenient and natural language in which he can describe his problem. We believe that GEM2 is easier to use than any other PDE package; this is achieved without any pre-processor being required.

d) Extensibility. Equation (1) may represent a single or a set of coupled equations, with real or complex (scalar or vector valued) functions A, B, C, g, f.

GEM2 is written in mode-independent form; the underlying field is represented via a predefined mode scal, the choice of this mode determining the class of equations covered:

Class	mode scal;
Single real PDE	real
Single complex PDE	compl
Coupled real PDES	ref [ ] real
Coupled complex PDES	ref [ ] compl

The body of the code is written in terms of the mode scal; the four versions differ only in the provision of a small (≤ 100 lines) prelude containing declarations of the primitive data type (e.g. mode scal = ref [ ] real) and of a few primitive operators on objects of these modes. These features of GEM 2 offer an interesting

demonstration of the use of an extensible language in a non-trivial mathematical software environment; we estimate that they have cut the development costs by a factor of three to five.

2. HOW

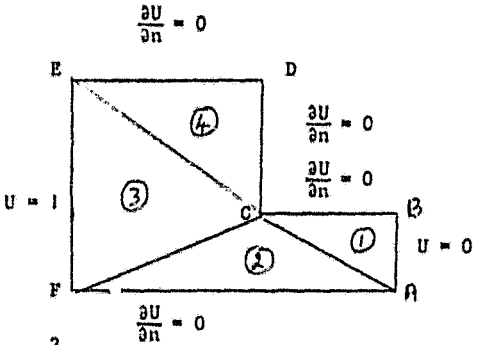
GEM 2 implements the Global Element method [1,2]; that is, the given region is subdivided into (a few) elements; these are mapped onto a standard region (a square) and a high-order orthogonal polynomial approximation used within each mapped element. The objects required to implement the method include sides (of elements); elements; maps; differential equation coefficients; differential equations; boundary conditions. Each of these has a corresponding mode declaration in GEM 2. Sample (skeleton) mode declarations include:

```
mode sidefunc = struct (Union (proc (real) real,
proc [ ] real, real) real) func, ref [ ] real
params);
mode side = struct [1:3] real position, int
mode, [1:2] sidefunc shape);
mode element = struct (ref [ ] int side numbers,
variety of other stuff);
```

The effect of these declarations is that elements are described in terms of their edges; and the user has a fairly wide choice of modes of description of a side: Cartesian, polar, or parametric coordinates, with or without parameters. As indicated, the actual structures contain a variety of other useful but more technical bits and pieces; we found it extremely useful during development to be able to add another field to the defining structure and have the appropriate information propagate painlessly through the entire program. Note also (for PASCAL addicts) that we keep procedures inside structures; we cannot imagine writing GEM 2 without this facility. We also make extremely free use of the ability to generate global space dynamically (the "heap"). Unions ("variant records") are used to yield a flexible interface for the user.

3. HOW SUCCESSFUL (was the decision to use Algol 68)? The answer to this has three aspects:

Q Did it help implementation? A: We are totally hooked.
Q How about runtime efficiency? A: Depends on your implementation; on our computer, (ICL 1906S) Algol 68 runs about as fast as FORTRAN.
Q How easy to use is GEM 2 really? A: Try to judge from the following complete program.



Solve $\nabla^2 U = 0$ in an L-shaped region with boundary conditions as shown.

User program:

```
begin problem L shape; describe problem (L shape,
4, 9);
real ab,bc,cd,de,ed; read ((ab,bc,cd,de));
af: = ab + cd; fa: = bc + de;
print ((newline, "ab bc cd de", ab,bc,cd,de));
[ ] real origin: = (bc - fa, -ab.0,0) 'c' global,
local origin at corner c f 'c';
point a = (fa,0.0), b = (fa,ab), c = (fa - bc,ab),
d = (fa - bc, af), e = (0, af), f = (0,0,0,0);
Triangle (1, (5,6,7), c,b,a), origin, 1,2/3,
false);
Triangle (2, (7,1,8), c,a,f, origin, 1,2/3, false);
Triangle (3, (8,2,9),c,f,e, origin, 1,2/3, false);
Triangle (4, (9,3,4), c,e,d, origin, 1,2/3, false);
Internal ((7,8,9)); Neumann ((1,3,4,5));
Dirichlet (6, constant 1.0);
Dirichlet (6, constant 0.0);
for n from 3 to 9 do
solve (n, 1,0.1,);
Tabulate ( noparams current value, null defunction,
6,6)
od
end
```

A Portable File System*

David R. Hanson

Department of Computer Science, The University of Arizona
Tucson, Arizona 85721

1. Introduction

Input/output is one of the most machine-dependent aspects of programming, especially for portable software. The large range of i/o and file system capabilities among existing computer systems makes it extremely difficult to avoid idiosyncratic problems in even the most carefully engineered portable systems. A typical solution to this dilemma is to use the 'standard' i/o and file primitives defined in the language in which the portable system is written [tan78]. In Fortran, for example, it is common practice to use only the forms of the i/o statements defined in the ANSI standard, and to use verifiers that aid in the detection of non-standard constructs [ryd74]. Another common approach is to define a small set of relatively low-level i/o routines that can be easily implemented and can model the capabilities of most commonly available file systems. By funnelling all i/o through these routines, portability problems are isolated in their machine-dependent implementation. The software described in [ker76] is evidence of the success of this approach.

A problem with these traditional approaches is they invariably sacrifice capability and efficiency for portability. Designs based on these approaches tend to have only the 'lowest common denominator' in capabilities of the intended host systems, such as sequential i/o to character files of restricted names. Enhancements may of course be added, but at the expense of an increase in implementation complexity and a reduction in portability.

The heart of the problem with traditional approaches to portable i/o systems lies in their attempt to manipulate highly machine-dependent objects--host machine files and file names. This paper describes a portable file system that makes files and their names machine-independent. The most important advantage of this approach is that i/o is not limited by the target systems. For example, capabilities such as random access, multiple access, and automatic expansion of files, which are absent in some commercial operating systems, are provided by the portable file system.

The portable file system—PFS for short—is the combination of a portable file directory system [han80a, han81] and a portable i/o system [han80b]. It provides machine-independent files and file names, a hierarchical directory structure in which to organize files, and a set of directory manipulation and i/o primitives. The directory structure and primitives are similar to the structure and primitives of the UNIX [rit74] file system. The PFS is, in large part, a portable implementation of the UNIX file system. It is packaged as a set of Ratfor [ker75] (and hence Fortran) functions and subroutines, which is loaded with the program or system that uses it. The implementation techniques are similar to those used in UNIX [tho78] and are described in [han80a] and [han80b].

2. Directories

The directory structure in the PFS is a rooted tree structure in which the leaves are files or directories and the nodes are directories. A directory is simply a list of files and directories. The root of the tree is denoted by /, and files and directories are denoted by their 'path', which specifies their absolute position in the tree.

A path is composed of the names of the nodes on the path from the root to the desired file or directory. The path components are separated by slashes, e.g. **/source/pfs/alloc.rat**. The names '.' and '..' refer, respectively, to the directory itself, and to its immediate ancestor. These names may be used as path components, providing a explicit means of using the structural properties of the tree. If a file name does not begin with /, it is taken to be rooted at the 'current directory'. For example, if the current directory is at **/source/pfs**, the name **alloc.rat** refers to **/source/pfs/alloc.rat**. Files and directories are equivalent with the exception that directories cannot be written by the user.

The primitives that deal exclusively with the directory structure are summarized in Table I.

Table I. PFS Directory Primitives

chdir(name)	change current directory to name
link(name1,name2)	make a link to name1 named name2
mkdir(name)	make a directory named name
rmdir(name)	remove directory named name
stat(name,array)	return information about file name

The current directory is changed by **chdir**. **link** establishes alternate names for a file. Directories are created by **mkdir** and, once empty, are deleted by **rmdir**. Information about a file (or directory), such as its size and date of creation, is returned by **stat**.

3. Primitives

A PFS *file* is similar to a file in UNIX and may be thought of as a finite sequence of characters or bytes. The PFS is insensitive to the range of byte values, so it can accommodate both 'binary' and 'character' files. Primitives are provided to create, delete, and open files, and to read and write characters anywhere within a file. Files are as large as is necessary to accommodate what is written to them, but are otherwise featureless. The basic primitives are summarized in Table II. Most primitives return a value indicating the success or failure of the operation.

Table II. PFS I/O Primitives

fd = fopen(name,mode)	open file name
fd = fcreate(name,mode)	create and open file name
fclose(fd)	close a file
n = fread(buffer,count,fd)	read from a file
n = fwrite(buffer,count,fd)	write to a file
pos = fpos(offset,type,fd)	position 'i/o pointer'
fremove(name)	delete file name

Existing files are opened for i/o by **fopen**. The argument **name** is the name of the file and **mode** is **READ**, **WRITE**, **READWRITE**, or **APPEND** and indicates 'how' the file is to be accessed. If the file exists, **fopen** returns a 'file descriptor', which may be thought of as a 'handle' that is used to access an opened file. Descriptors are similar in concept to channel numbers or Fortran unit numbers. Their values are never inspected explicitly but are passed to other primitives to indicate the opened file on which they should operate.

New files are created by **fcreate**, which creates the named file and opens it as if **fopen** had been called. If the file already exists, it is truncated to zero length and opened.

Opened files are closed by **fclose(fd)**.

Data transfer to and from opened files is performed by **fread** and **fwrite**. **fread** reads up to **count** characters from the opened file indicated by the file descriptor **fd** into **buffer**. It returns the number of characters actually read, which may be 0 when the end of the file is reached. **fwrite** writes **count** characters from **buffer** to the file indicated by **fd**. Writing beyond the current size of the file is permitted, and the file is automatically extended to accommodate what is written to it. For symmetry with **fread**, **fwrite** returns the number of characters actually written.

An 'i/o pointer' is associated with each opened file and is advanced by **fread** and **fwrite**. It can be repositioned by **fpos** according to the values of **offset** and **type**. If **type** is 0, **offset** specifies a position relative to the beginning of the file; if **type** is 1, **offset** specifies a position relative to the end of the file; and if **type** is 2, **offset** specifies a position relative to the current positon of the file. **fpos** returns the previous position of the file.

Files are deleted by **fremove**. Deletion of opened files and files with aliases is permitted; the file is actually deleted upon the removal of the last reference to it. After deletion, all space occupied for the file is available for reuse.

4. Conclusions

The portable file system provides a machine-independent concept of file and i o primitives that offer greater flexibility than is found in many operating systems. It is typically more efficient than the traditional approaches to portable i o systems. For example, measurements on a DEC-10 and Cyber 175 show a 25-35 percent improvement over Fortran i o for sequential character files.

Perhaps the best characterization of PFS is an abstract data type 'file'. It provides a data structure, file, and a set of operations on that structure. This characterization clarifies the important difference between the PFS approach and traditional approaches, which attempt to provide a set of operations on unspecified and highly machine-dependent data structures - host machine files.

References

[han80a]
Hanson, D. R. A Portable File Directory System, *Software - Practice & Experience 10*, 8 (Aug. 1980), 623-634.

[han80b]
Hanson, D. R. A Portable Input Output System, Tech. Report 79-17a, Dept. of Computer Science, The University of Arizona, Tucson, Nov. 1980.

[han81]
Hanson, D. R. Algorithm 568: PDS - A Portable Directory System, *ACM TOPLAS 3*, 2 (Apr. 1981), 162-167.

[ker75]
Kernighan, B. W. Ratfor - A Preprocessor for a Rational Fortran, *Software - Practice & Experience 5*, 4 (Dec. 1975), 396-406.

[ker76]
Kernighan, B. W. and Plauger, P. J. *Software Tools*, Addison-Wesley, Reading, MA. 1976.

[rit74]
Ritchie, D. M. and Thompson, K. The UNIX Timesharing System, *Comm. ACM 17*, 6 (Jul. 1974), 365-375.

[ryd74]
Ryder, B. G. The PFORT Verifier, *Software - Practice and Experience 4*, 4 (Dec. 1974), 359-377.

[tan78]
Tannenbaum, A. S., Klint, P. and Bohm, W. Guidelines for Program Portability, *Software - Practice and Experience 8*, 6 (Nov. 1978), 681-698.

[tho78]
Thompson, K. UNIX Implementation, *Bell System Tech. J. 57*, 6 (Jul. 1978), 1931-1946.

*This work was supported by the National Science Foundation under Grant MCS-7802545.

TOOLPACK- A Collection of Tools for Mathematical Software*

Leon Osterweil
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

Introduction

TOOLPACK is a cooperative project involving researchers at Argonne National Laboratories, Bell Telephone Laboratories, International Mathematical and Statistical Libraries, Inc., Jet Propulsion Laboratory, Numerical Algorithms Group, Ltd., Purdue University, University of California at Santa Barbara and University of Colorado. The project is being funded by the Dept. of Energy and the National Science Foundation, as well as the participating institutions. TOOLPACK has as its objective the conveyance of strong comprehensive tool support to programmers who are writing, testing, transporting or analyzing mathematical software. Hence it must provide strong support for documentation, testing, and verification, as well as such code creation activities as editing.

A wide variety of tools will be built and adapted to support these activities. It is expected that these tools will be distributed as stand-alone entities. There is, however, considerable support and sentiment in favor of creating an integrated collection of the tools as well. This paper presents a brief overview of the design of and plans for this integrated collection. The following summary is extracted from [Oste 81], wherein additional details can be found.

Before commencing with description of the design, it is important to enunciate the following basic assumptions:

1. The mathematical software whose writing, testing, and analysis is to be supported by TOOLPACK is to be written in a dialect of Fortran 77, which shall be carefully chosen to span the needs of as broad and numerous a user community as is practical.

2. TOOLPACK is to be designed to provide cost effective support for the production by up to 3 programmers of programs whose length is up to 5000 lines of source text. TOOLPACK may be less effective in supporting larger projects.

3. TOOLPACK is to be designed to provide cost effective support for the analysis and transporting of programs whose length is up to 10,000 lines of source text. TOOLPACK may be less effective in supporting larger projects.

4. TOOLPACK will support users working in either batch or interactive mode, but may offer stronger more flexible support to interactive users.

A. Overview

A primary motivating goal of the TOOLPACK integrated tool collection design is that user support be supplied in as direct and painless a fashion as is feasible. In particular, the design attempts to relieve the user of having to understand the natures and idiosyncrasies of individual TOOLPACK tools. It also relieves the user of the burden of having to combine or coordinate these tools. Instead the design encourages the user to express his needs in terms of the requirements of his own software job. The TOOLPACK support system is designed to then ascertain which tools are necessary, properly configure those tools, and present the results of using the tools to the user in a convenient form.

The design encourages the user to think of TOOLPACK as an energetic, reasonably bright assistant, capable of answering questions, performing menial but onerous tasks and storing and retrieving important bodies of data. The aim of this is to make humans more effective in creating, documenting, testing and verifying program code.

In order to reach this view, the user should think of TOOLPACK as a vehicle for establishing and maintaining a file system containing all information important to the user, and using that file system to both furnish input to needed tools and capture the output of those tools. Clearly, such a file system is potentially quite large and is to contain a diversity of stored entities. Source code modules would certainly reside in the file system, but so would such more arcane entities as token lists, and flowgraph annotations. In order to keep TOOLPACK's user image as straightforward as possible the design proposes that most file system management be done automatically and internally to the TOOLPACK system, out of the sight and sphere of responsibility of the user. The user may create, delete, alter and rename these entities. The user may, however, also manipulate these entities with a set of commands which selectively and automatically configure and actuate the TOOLPACK tool ensemble. The commands are designed to be easy to understand and use. They borrow heavily on the terminology used by programmers in creating and testing code, and conceal the sometimes considerable tool mechanisms needed to effect the results desired by the user.

B. User Visible File System Entities

In order to encourage and facilitate the preceding view of TOOLPACK, the system will support the naming, storage, retrieval, editing and manipulation of the following classes of entities, which should be considered to be the basic objects of TOOLPACK:

1. Program units:

A TOOLPACK program unit (PU) is the same as a Fortran program unit, except that TOOLPACK will require a number of representations of the program unit other than the source code (e.g., the corresponding token list and parse tree).

2 Execution Units:

Any set of TOOLPACK program units which the user chooses to designate, can be grouped into a TOOLPACK execution unit (EU).

3. Test Data Collections:

A TOOLPACK test data collection (TDC) is a collection of test data sets to be used in exercising one or more TOOLPACK execution units.

4. Options Packets:

A TOOLPACK options packet (OP) is a set of directives specifying which of the many anticipated options are to be in force for a particular invocation of a particular TOOLPACK tool.

C. The TOOLPACK Command Language

The exact syntax for the TOOLPACK integrated tool collection command language has not been established and is still under study. Currently we are in a position, however, to specify much of the semantic content of this language.

The proposed TOOLPACK command set seems to divide logically into four parts: file system management commands, edit (synthesis) commands, tool application (analysis) commands, and perusal commands.

1. File System Manipulation

These will facilitate the creation, deletion, renaming and general maintenance of TOOLPACK files.

2. Edit (synthesis)

These commands would summon special purpose editors designed to facilitate the manipulation, examination and alteration of the contents of the various TOOLPACK file system entities.

3. Tool Invocation (Analysis)

These commands invoke the functions which are at the heart of the reason for the TOOLPACK project - namely the facilitation of documentation, testing, verification, transportation, and source program entry. Consequently, great pains are being taken to make them easy to understand and use. In an important sense, the rest of the TOOLPACK command language has been designed so as to make these tool invocations straightforward.

a. FORMAT

Invocation of this command causes a named program unit to be taken as input to the TOOLPACK formatting tool.

b. STRUCTURE

Invocation of this command causes a named program unit to be taken as input to the TOOLPACK structurer.

c. ANALYZE

Invocation of this command results in the static analysis of the entity named. If the entity is a program unit, then single unit analysis will be performed. If the entity is an execution unit, then each program unit will be analyzed individually and integration analysis will also be performed.

An options packet may be specified by the user. This packet will enable the user to specify a level of thoroughness which will cause analysis to go as far as the lexical level the syntactic level, the static semantic level or the data flow level. If this specification is omitted, the TOOLPACK system will select a default option (probably full data flow analysis).

The results of this analysis will be placed into an entity-attribute-relational data base which will then be available for perusal by a browsing subsystem or for use as the basis for report generation tools whose goal would be the creation of superior documentation.

d. EXECUTE TEST

Invocation of this command results in the dynamic test execution of a collection of test data sets by a specified execution unit. The test data sets comprising the test data collection are fed into the execution module derived from the execution unit one at a time, with the results of each execution being used to build an execution history data base. This data base would be used to supply answers to user-posed questions as well as reports needed for documentation purposes.

The user may optionally specify a test options packet whose purpose is to select and specify which of the numerous execution monitoring options are to be employed during the test runs. The power and flexibility of the dynamic test monitoring system is to be considerable (see [Feib 81]). This is deemed to be necessary, but is also considered to be a serious problem, in that a casual or novice user may be intimidated by the variety of available choices. Hence it is proposed that a set of standard Test Option Packets (TOP's) be prepared by the builders of the dynamic test monitoring system and stored permanently in the TOOLPACK file system. Users could select from among these, tailor them to individual needs by using the TOP editor, or create their own TOP's from scratch. One of the standard TOP's would be configured to be the default TOP, enabling the user to do useful dynamic testing without needing to specify any TOP.

4 Perusal

TOOLPACK will ultimately contain tools to facilitate the examination of the various entities in the TOOLPACK file system. This abstract has already mentioned various special purpose editors, part of whose purpose will be to facilitate examination of the user-named file system entities (e.g., the PU source text, EU's, OP's and TDC's). A different sort of tool is desirable for use in perusing the output of the static analysis and dynamic testing tools. As already noted, these tools will produce as output sets of analytic and diagnostic packets which are most profitably viewed as relational data bases. Tools for effectively browsing these data bases could be specifically constructed to efficiently scan these data bases for answers to expected queries. Existing text editors will probably serve as primitive forerunners of these tools in early releases of TOOLPACK.

D. Implementation Plans and Schedule

The TOOLPACK integrated tool collection is scheduled for public release in January, 1983. Preliminary releases to test sites will take place during 1982. Individual tools will be made available intermittently.

REFERENCES

[Feib 81] J. Feiber, R. N. Taylor, L. J. Osterweil, "Newton--A Dynamic Program Analysis Tool Capabilities Specification," Tech. Report #CU-CS-200-81, Dept. of Computer Science, University of Colo., Boulder, Colo.

[Oste 81] L. J. Osterweil, "Draft TOOLPACK Architectural Design," technical memorandum, University of Colorado at Boulder, Dept. of Computer Science, March 1981; available from Applied Mathematics Div., Argonne National Lab., Argonne, Ill.

*This work supported by NSF grant number MCS8000017 and DOE grant number DE-AC02-80ER10718.

SOFTOOL 80™
A METHODOLOGY AND INTEGRATED COLLECTION OF TOOLS FOR SOFTWARE MANAGEMENT, DEVELOPMENT, AND MAINTENANCE

by

EDWARD MEHLSCHAU
SOFTOOL CORPORATION
340 SOUTH KELLOGG AVE.
GOLETA, CALIF. 93117

SOFTOOL 80™ is a methodology and an integrated collection of tools that addresses the entire software development process. Release I of SOFTOOL 80™ addresses the programming phase of software development (that is, the portion of the software development life cycle that begins after a detailed design document has been generated and continues to the point where a complete, deliverable software product has been produced). Forthcoming releases of SOFTOOL 80™ address the remaining portions of the software life cycle. This paper outlines a subset of SOFTOOL 80™ Release I that provides a powerful environment for the development of mathematical software.

SELECTED TOOLS

The AUDITOR:

The AUDITOR is a software product that automatically documents Fortran programs for deviations from a user-defined standard, poor programming practices, and non-portable code.

The American National Standards Institute (ANSI) definition of Fortran is used by the AUDITOR as a baseline or default standard. A user can defin any standard desired by instructing the AUDITOR to allow extensions to the ANSI definition. The AUDITOR incorporates a powerful compile-time diagnostic capability equal or superior to that of the best commercial compilers. The messages generated can be classified into six categories:

1. Error Messages -- definite violations of the standard.
2. Warning Messages -- potential violations of the standard.
3. Portability Messages -- program transferability problems.
4. Documentation Messages -- program documentation.
5. Confirmation Messages -- completion of analysis indication.
6. System Messages -- internal system information.

The INSTRUMENTERS:

Under SOFTOOL 80™, Fortran programs are instrumented for three different purposes: tracing, testing, and optimization. The TRACING INSTRUMENTERS generate profiles that indicate the levels (i.e., routine, statement) and the path traversed by the program flow of control during execution. The TESTING INSTRUMENTERS generate profiles that indicate the coverage and effectiveness of test runs. The OPTIMIZATION INSTRUMENTERS generate profiles that pinpoint the most time consuming sections of code in a system.

The INTERFACE DOCUMENTER:

The purpose of the INTERFACE DOCUMENTER is to generate clear, up-to-date, and complete documentation of all interfaces between object modules. Reports produced include:

1. A complete cross reference of all symbols defined or referenced in the object modules which were input to the tool.
2. A list of all symbols referenced but not included in the modules input to the tool for analysis.
3. A list of all symbols not referenced by any of the modules processed by the tool.
4. A summary of the interconnections and the lengths of all the object modules processed by the tool.
5. A summary of all recursion that occured in the modules input to the tool.
6. Two optional reports known as explosion and implosion. Explosion displays the hierarchical structure of all symbols referenced, directly or indirectly, by a user specified symbol. In other words, explosion displays the subsystem referenced by the specified symbol. Implosion, on the other hand, shows the hierarchical structure of all symbols that reference the given symbol.

The PRODUCTIVITY LIBRARY:

The PRODUCTIVITY LIBRARY allows the user to interactively define high level data structures and provides a set of primitives for manipulating and searching the defined data structures from the user program. Range checking and dynamic flow analysis can also be specified for the user-defined data structures. With range checking the user may specify values that data structure items are allowed to assume. Dynamic flow analysis allows the user to specify allowable flow transitions for a given item in a data structure.

SUMMARY

The objective of this paper is to outline a subset of the tools available under SOFTOOL 80™ Release I. These tools act as a fine sieve, forming an environment for the development of mathematical software that eliminates a large class of potential problems. These problems would otherwise manifest themselves at a later time as "bugs".

The tools available under SOFTOOL 80™ Release I that were not described in this paper include:

- short-hand language
- structured language
- source code documenters
- dynamic memory managers
- tutorials

A Support Environment for Software Tools

Fred T. Krogh and W. Van Snyder
Jet Propulsion Laboratory

1. Introduction.

We summarize here a support environment useful for the development of software tools, described more fully in [1]. There are five components: I/O Primitives, filing system, working storage manager, symbol table manager and lexer / parser. The working storage manager, filing system and I/O primitives collaborate to provide a segmented virtually addressable storage access method similar to the Multics environment: a user program can

> "directly address just those items it needs from the extensive on-line data files, so that each reference to such items can (in the logical sense) be a single operation. The actual reference need not be preceded by an input/output system request to input a (partial copy of the) file, nor be followed by an input / output system request to output the altered information to its original location." [2]

2. I/O Primitives.

The primary function of the I/O primitives is to provide a transportable and efficient interface to the idiosyncrasies of various operating systems. Software tools require services provided by sequential character input and output devices such as keyboards and printers, and sequential and direct access storage devices.

To provide efficiency, access to sequential storage devices is asynchronous and double buffered; the details of the provision of such efficiency are hidden inside the primitives charged with providing such functions. Operating system services providing synchronous and asynchronous access to direct access storage devices, and services required for synchronization of the primitives with operation of the devices are used to provide efficient direct access.

3. Filing System.

The filing system is responsible for storing and organizing representations of program units, and tables required by higher level tools. The filing system allows the user to express relations between objects. The user, for example, is allowed (encouraged!) to declare that a program consists of a related collection of subprograms, etc. The operating environment of the filing system is provided by the I/O primitives and the working storage manager.

A simple model of the filing system is a road map (of one way roads). An object is accessed by specifying an ordered list of "roads" to be traversed to proceed from a standard starting place to the desired object. A collection of objects is denoted by a path to some part of the road map from which the collection, but only the collection, may be reached. The road map is represented as a directed graph. Roads are represented by edges in the graph. Intersections of roads, and objects in the filing system, are represented by vertices in the graph.

4. Working Storage Manager.

The working storage manager operates in the environment provided by the I/O primitives and the filing system, and in turn provides important features of the environment needed by those tools. The primary function of the working storage manager is to provide space to other tools, and recover that space when it is no longer needed. Any significant restriction of the amount of space available to a high level tool is not acceptable. The working storage manager is therefore also responsible for providing the illusion of unlimited storage capacity.

Providing the illusion of unlimited storage capacity is usually accomplished by the use of a heirarchical storage system, where frequently used data are retained in main storage, and infrequently used data are retained on secondary storage. There is a correspondence between addressing spaces managed by the working storage manager, and objects in the filing system. The filing system is used for the secondary storage medium required to provide the illusion of unlimited storage capacity. In return, information in the filing system is accessed as though it were stored in the main memory, rather than by the direct use of I/O primitives. This mechanism imposes very little overhead.

5. Symbol Table Manager.

The symbol table is logically divided into three parts: a local symbol table for each program unit, describing local characteristics of variables, constants, labels, control structures and other objects; a global symbol table for each program unit and collection of program units describing interface information for each program unit and the relations between calling and called subprograms; and a language symbol table describing such objects as keywords, intrinsic functions, and spelled operators.

Access to objects in the symbol tables is provided by functions, which should be expanded in line rather than called. These functions assume the object to be accessed is in main memory - an illusion provided by the working storage manager.

6. Lexer/Parser.

The lexer and parser are usually assumed to be separate tools. But a Fortran lexer may be made significantly simpler if the parser is available to give it advice. The integration of the lexer and parser does not impose an unbearable storage penalty on applications needing only lexical analysis.

The lexer / parser is controlled by tables, stored in the standard filing system, and accessed by the working storage manager. This allows one lexer / parser to accept several dialects of Fortran, preprocessor dialects, or commands, depending on the tables used to control its operation.

All input to and output from the lexer / parser is accomplished by the working storage manager. If the source text is not an object in the filing system, the working storage manager provides the illusion that it is. At the conclusion of analysis, the source text may be retained in the filing system if desired, even if it was not originally an object in the filing system. All other products of the analysis may also be stored in the filing system as desired.

7. References.

1. Fred T. Krogh, W. Van Snyder, Section 366 Computing Memorandum 476, Jet Propulsion Laboratory (May 1981).

2. Elliott I. Organick, "The Multics System: An Examination of its Structure," MIT Press, Cambridge, MA (1972) p 1.

This work represents the results of one phase of research carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract No. NAS 7-100, sponsored by the National Aeronautics and Space Administration.

A Method for Constructing Preprocessors

Daniel I. Boley, William D. Gropp, and Marvin M. Theimer

Many programming and command languages provide only limited features and painful user-interfaces (e.g. Fortran and JCL). Preprocessors are programs that can extend and reformat such languages using translation techniques akin to compilers. They can be valuable tools in the development of mathematical software. However, writing a preprocessor can be a time consuming task.

We present a method for quickly constructing a preprocessor from a formal description of the language to be translated. We first discuss the properties to expect from a good preprocessor. We then describe how we used existing techniques to achieve these goals. We present an example of a FORTRAN preprocessor constructed by our system.

A good preprocessor should have the following properties.
- It must be easy to write the preprocessor correctly, and it should perform reasonably efficiently.
- Error recovery and reporting should be part of the formalism where possible, not just an addendum.
- It should be easy to modify the language accepted by the preprocessor.
- There must be the capability to escape back to the base language.
- The preprocessor will have to be portable.
- The generated output code should be reasonably efficient and readable.

To achieve these goals, we make the following recomendations. First, a pseudo-compiler approach should be taken. This provides a modular framework employing a scanner module to recognize input tokens or symbols, a parser module to recognize the syntax of the language, a semantic package to generate the desired output, and an error package to handle error recovery.

Second, the input language should be formally specified using a syntax grammar with attached semantic actions. This provides a concise, easily modifiable description of the language which fits nicely into the pseudo-compiler approach.

Third, the above choices along with the type of capabilities usually desired imply that the most difficult parts of the preprocessor can be written using well-established tools and techniques.

Finally, use the simplest approach possible. Unless available as a package, a more powerful technique may yield very little advantage.

As an example of an actual implementation of these ideas we present the choices made for our implementation of an extended Ratfor preprocessor for Fortran.

Scanner:
Our scanner is "hand-built". We considered using a scanner generator package to automatically generate a scanner routine; however, a suitable package was not easily available. Since most preprocessor scanners are very simple, it is often true that a hand-built version is sufficient and can be constructed quickly.

Parser:
A table-driven parser was built. This kind of parser is small, fast, and allows easy modification of the input grammar. Of the two main techniques in use (LL and LR) we chose the less powerful LL one. The reasons for this were that an LL parser is easier to create and makes certain types of error recovery easier. The fact that this technique cannot handle as large a class of languages as LR was of no real concern since it can handle the type of input languages that most preprocessors are oriented towards.

Semantic package:
This section of the preprocessor was again hand-tailored to the application. This is primarily because we know of no practical technique which allows a formalization of this area. However, by using a scheme of grammar attached semantic actions, it is possible to divide up the tasks of the semantic package into small, manageable pieces.

Error package:
By using an LL parsing technique it is possible to automatically detect, report, and recover from syntactic errors in an input program. Unfortunately, semantic errors in a program must still be handled in an ad-hoc fashion (this is also true for an LR parser).

In summary, our basic attitude was to follow a pseudo-compiler approach using whatever tools or techniques were available or easy to implement.

Currently we have an extended version of the original Ratfor preprocessor of Kernighan and Plauger implemented using our approach. The final version will include the following extensions:
- data structures, foremost of which are records and variable-origin arrays.
- an output code format that is as readable as possible.
- various special operators, including array-based operators.
- execution time profiler to allow easy invocation of timing statements.
- free-format I/O.

Some other preprocessors that could be constructed with this method are:
- a preprocessor for the IBM VM operating system command language.
- a PDE language on top of Fortran. For example, this language could be oriented towards convenient specification of mesh refinement programs.
- an array language, perhaps oriented toward optimizing code produced for a specific vector machine. (Preprocessed constructs allow for painless insertion of assembly-level language.)
- a front-end interface for large mathematical software packages.

A Mutation Analysis of Numerical Software

M.A.Hennell, I.J.Riddell and M.R.Woodward
Dept. Computational & Statistical Science
University of Liverpool
Liverpool L69 3BX, U.K.

One convincing way to demonstrate that a particular program contains no errors is to generate systematically all the possibilities and then demonstrate that the test data can detect them all! A more realistic approach is advocated by De Millo [1] who has constructed an interpretive system to generate many simple errors, e.g. errors in individual lexemes, in such a way that each error is effectively inserted in its own copy of the original program. These copies with seeded errors are known as "mutant programs". The idea is then to construct test data which kills all the mutant programs by showing incorrect output data. Mutants which are equivalent to the original programs will, of course, remain live.

There are a number of problems with this approach, one of which is the enormous number of mutants which might be generated. For instance, for a thirty line program, fifteen hundred mutants is a realistic possibility and the number of mutants grows roughly with the square of the length of the program. Therefore, a specialised system (such as that of [1]) is required for practical use.

However, there are various subsets of mutants, not necessarily exclusive, which may be of particular interest and for these subsets the number of generated mutants may be manageable in a conventional compile and run system. The authors have built a mutation system which utilises a standard editor together with compilers and run-time systems for FORTRAN, COBOL and ALGOL68.

In this paper we describe our experiences using the FORTRAN version of the mutation system to analyse the adequacy of the test data sets for a number of numerical routines.

We have analysed three subsets of mutants. The first subset is that arising from the relational operators. From a statistical analysis [2] of numerical software we know that, on average, there will be eight relational operators per routine, and, since each relational operator can be replaced by five alternative relational operators, we obtain a total of about forty mutants per routine. The subset is therefore manageable and is also of interest, since testing the predicates is one of the more difficult aspects of path testing, particularly if the predicates are complex.

The test data adequacy of [4] is defined as

$$A1 = \frac{\text{number of dead mutants}}{\text{total number of mutants}}$$

We have measured the test data adequacy for a number of NAG routines and the average value of A1 for these routines is 0.66 (with a standard deviation of 0.22). Unfortunately it is extremely difficult to distinguish the equivalent programs from the remaining live mutants, so the adequacy of the test data may be substantially higher than this measure indicates. Our results show that conventional test data, which does not achieve particularly good coverage in terms of ,say, the TER metrics of [3], nevertheless does tend to detect simple errors in the relational operators.

The second subset is the control flow mutants. Mutants are created which affect the flow of control, such entities are:

a) negating the relational operators and logical values
b) replacing labels in each computed GOTO and arithmetic IF by permutations of the other labels
c) incrementing the upper index of DO loops.

Test data to detect these errors will in general need to achieve coverage ratios close to TER2 = 1.0, although TER1 = 1.0 does not need to be satisfied. This is due to the logical IFs, since, for example, only one leg of an IF-THEN-ELSE construct would need to be executed to detect a negated predicate. The results for this subset yield an average A2 ratio of 0.57 (standard deviation 0.27) for the routines tested. A more detailed inspection of the results, however, reveals that this ratio is boosted by a high percentage of dead mutants in category a) (averaging 80%) compared to the relatively low percentages of dead mutants in categories b) and c) (averaging about 40% in each case). Hence, these results indicate that, on the whole, errors in the arithmetic IFs, computed GOTOs and DO loops are not easily found with conventional test data sets.

The third set are the relational operators and arithmetic IFs which determine the inclusive boundaries of the input data domains. These are the relational operators involving equality, i.e. .EQ., .NE., .LE., .GE., and also the zero case of the arithmetic IF. Results obtained for this subset lead to an average A3 ratio of 0.64 (standard deviation 0.22) for the routines tested. Again the relatively high percentage of dead mutants in the relational operators (approximately 65%) dominates the percentage of dead mutants in the arithmetic IFs (approximately 26%).This indicates that providing test data for these domain boundaries is not a common occurrence.

From this work we can conclude that a limited mutation analysis, concentrating on particular subsets which demonstrate a specific testing strategy, is a worthwhile activity. The cost of the exercise, being of the order of fifty runs per numerical routine, is acceptable, although it must be noted that this cost rises if the initial test data adequacy is low. Our experiments show that when mutants are dead the differences between correct and incorrect output tend to appear in the first 30% of the output. This possibly shows that testers tend to put special cases and other tricky data first and more general cases last. The effect, however, is beneficial to an automatic comparator. With the provision of a specialised mutation system a dramatic cost reduction can be obtained or a wider class of mutant be considered.

[1] De Millo, R.A., Lipton, R.J. and Sayward, F.G.,
Hints on test data selection: help for the practicing programmer.
Computer,Vol 11,No.4,pp.34-41.April 1978.

[2] Hennell, M.A. and Prudom, A.,
A Static Analysis of the NAG Library.
IEEE Transactions on Software Engineering, Vol 6, No. 4, pp.329-333, July 1980.

[3] Woodward, M.R., Hedley, D. and Hennell, M.A.,
Experience with Path Analysis and Testing of Programs.
IEEE Transactions on Software Engineering, Vol 6, No. 3, pp.278-286, May 1980.

[4] Burns, J.E.,
Stability of Test Data From Program Mutation
Digest for the Workshop on Software Testing & Test Documentation, Ft.Lauderdale, Florida, Dec. 18-20, 1978.

A COURSE ON MATHEMATICAL SOFTWARE

A. K. Cline

University of Texas at Austin

During the four academic years ending in 1981, a one semester course in mathematical software construction has been offered by the Computer Sciences Department at the University of Texas at Austin. Although originally offered at a graduate level, a modified version has been offered three times for undergraduate students. Those electing the class include students interested in scientific programming as a profession as well as mathematics, engineering, and science students who seek to improve their programming skills.

Purpose:
The course purposes are to aquaint students with advanced programming topics and the theory of quality software construction and to allow students to gain practical experience working on a group project involving organizing, coding, and testing a package of mathematical software subprograms.

Prerequisite:
Students are first required to have an interest in programming for technical applications. A basic knowledge of FORTRAN is assumed as well as mathematics at the calculus level. A course in numerical methods is not required but may be so in the future since many examples in the lectures on programming methodology require familiarity with basic numerical algorithms such as Gaussian elimination and linear interpolation.

Text:
The lecture material in the course has been obtained from the author's experience and follows no published text. However, several short paperbacks are recommended for collateral reading:

Ledgard, H. F., Programming Proverbs for FORTRAN Programmers, Hayden, 1975.

Kreitzberg, C. B., and B. Shneiderman, The Elements of FORTRAN Style, Harcourt, brace, Jovanovich, 1972.

Format:
Until the final several weeks of the semester, the class meets three times per week for lectures. The lectures cover programming methodology and background material for the group project. These topics are described in greater detail below. Only occasional lectures are given at the end of the course as students complete their projects and meet individually or in groups with the instructor for criticism of their work.

Prior to the formation of the group, a short programming assignment is given to the class. This assignment has been the coding and testing of a simple module implementing a search in an ordered array. The algorithm suggested is a variant on binary search employing inverse linear interpolation. The purpose of this initial project is to gain experience on the local system (many students are new to file usage, editors, and interactive computing) and to give some indication of the students' relative programming capabilities. Another important indication is how energetic (alternatively, how procrastinating) each student performs the assignment. Although a grade on this assignment is not used in a final course grade determination, the students' grades as well as the length of time required to complete the assignment are made public. Students then organize themselves into three person groups. The performance on the first assignment has proved to be a very good predictor of a student's performance on the group project, and with this knowledge and the ability to form their own groups, we have attempted to minimize the common problem in group projects: an imbalance of attitude or abilities causes an imbalance in the work load.

The group project used in all four offerings of the course has been that of interpolating data specified on an irregular grid in the plane. The packages include modules to form a triangulation of the points in the plane and modules for interpolation based upon this triangulation. Several simplifying assumptions are made to avoid overconcern with mathematical details of the algorithm as opposed to the software implementation. These assumptions include ignoring the possibility of any colinearity of the points in the plane and, for the purpose of the smooth interpolation, that first partail derivative values are available from the user as well as function values. This project provides a good mix of mathematical and computer science problems.

A purpose of the course being to gain experience in software construction, and not necessarily the research level discovery of new algorithms; students are given a design of the package and brief descriptions of the algorithms. It is felt that this simulates well the situation in which a programmer is implementing the theory developed by another. It also allows lectures and class discussions on the algorithms with respect to their qualities.

A final examination is given with the intention of testing the lecture material on programming methodology which was not applied in the project.

Lecture Topics:
Initially a review of the usage of the local system is presented and a discussion of the project and the algorithms for its modules. The vocabulary of quality of software is then described: applicability, usability, efficiency, clarity, portability, modifiability, modularity, and flexability. With each characteristic examples are given and the importance (or unimportance) of problem areas is considered. The area of portability is explored in detail. The methodology for the testing of software and the design of software receive several lectures apiece (and are applied to the project). Several lectures on clever programming practices (e.g. decomposition of workspace, portable handling of mathematical constants, actions in error situations, timing of code) are given. Finally, several miscellaneous topics are covered: a brief introduction to scientific computer graphics including the use of the NCAR graphics package, various approaches to software documentation, and the 1977 FORTRAN standard.

MOVING SOFTWARE SYSTEMS TO A MINICOMPUTER

G. CIONI, A. MIOLA, A. TRUFFI
Istituto di Analisi dei Sistemi ed Informatica
Via Buonarroti 12, 00185 ROMA (ITALY)

In the last few years a well shaped phenomenon has been observed in the computer field: the cost of software production and maintenance is very high respect to the total cost of a computing system and it is still growing during this time.
Therefore it seems to be appropriate to use all the available software as much as possible respect to its portability, modularity and documentation. At the same time the new software is going to be designed and implemented according to the software engineering rules [1,2].
The use of a minicomputer, together with the decreasing use of time-sharing system on large computers, quite often presents the need to move software systems from a medium-large computer to a minicomputer.
This operation involves quite expensive transformations on the available software, while the quality of the final result, as far as the efficiency is concerned, is not foreseen.
In order to gain as much confidence as possible in the transfering process an "a priori" analysis is necessary and helpfull.
This paper actually presents the design and the implementation of procedures which supply informations on how to move software systems to a given minicomputer.
We will refer in this paper to software system organized as a library of programs, where a single program may also use other programs of the library as its subprograms. We call this organization System of Programs (SP). Generally a SP can also be seen as a collection of several moduli, each of which is itself a set of programs belonging to the library. If a modulus accomplishes a specific, well defined function, it is a subsystem of the given SP. We call this modulus Independent Subsystem (IS).
Therefore if a SP is represented as a tree an IS is a subtree of the given tree.
If the SP has been designed to be quite flexible and portable there is the possibility for the user to define the dimensions of the current data. Therefore a variable storage space is associated to each SP, together with a fixed storage space which is used for the local variables of the different programs. We denote the variable space by SPACE and the fixed space by LOCAL.
In the following we will refer to SP implemented in a programming language that uses static memory allocation. For instance FORTRAN.
Examples of SP to be considered are the symbolic and algebraic computation system SAC-1 [4,3] and the Harwell Subroutines Library [7]. Both these systems have been implemented in standard FORTRAN, they are portable, modular and very well documented.
In order to make a SP running on a computer which has less memory respect to the computer which the given SP was built for, one of the following techniques could be used:
- overlay of programs
- spliting of the SP into several IS
- reduction of the maximum size for SPACE
- programs segmentation.

Actually these techniques may also be used all together, or in any combination.
For each SP we have to figure out the total amount of words

$$M(SP) = COD(SP)+LOCAL(SP)+SPACE(SP)$$

to put our SP on the given computer, where COD(SP) is the number of words of the object code of SP. The quantity FIX(SP) = COD(SP)+LOCAL(SP) is fixed for the given SP, while SPACE(SP) is a variable quantity.
If we want use an overlay techniques we also need the tree structure representing all the calling relations between subroutines of SP, and again the total amount of words.
In general if M is the size of the given computer memory available to the SP we must check the quantity

$$D = M - FIX(SP)$$

In order to get all the need informations to test, we can certainly use the compiler of the SP implementation language, for instance FORTRAN, available on the given computer.
An automatic procedure has been designed to accomplish all the tests already described.

BIBLIOGRAPHY

[1] BAUER F.L. Editor: *Advanced Course of Software Engineering*, Lecture notes in Economics and Mathematical Systems, Springer-Verlag (1972).

[2] WIRTH N.: *Systematic Programming. An Introduction*, Prentice-Hall (1973).

[3] CIONI G., MIOLA A., TOZZOLI M.: *Symbolic and Algebraic Computation Systems on Minicomputers*, Proceedings of DECUS EUROPE Symposium (1980).

[4] COLLINS G.E.: *The SAC-1 System: An Introduction and Survey*, Proc. of SYMSAM II, ACM, Los Angeles (1971).

[5] ECKHOUSE R.H.: *Minicomputers Systems: Organization and Programming*, Prentice-Hall, N.Y.,(1975).

[6] GRIES D.: *Principles of Compiler Design*, Wiley, N.Y. (1971).

[7] HARWELL SUBROUTINES LIBRARY: *A Catalogue of Subroutines*, A.E.R.E. R7477 Supplements n. 1-2 (1970).

[8] JENSEN J., WIRTH N., PASCAL: *User Manual and Report*, Springer-Verlag (1975).

[9] KNUTH D.E.: *The Art of Computer Programming: Fundamental Algorithms*, Vol. I, Addison Wesley (1969).

[10] DONOVAN J.J., MADNICK S.E.: *Operating Systems, Mc GRAW HILL (1974)*.

[11] WEITZMANN C.: *Minicomputer Systems. Structure Implementation and Application*, Prentice-Hall, N.Y. (1974).

Tailoring Mathematical Software for the CRAY-1

David S. Dodson
John Gregg Lewis
William G. Poole, Jr.

Boeing Computer Services Company
Seattle, Washington

Any new computer brought into the stable of a major computer complex offers special challenges to those who are responsible for developing and maintaining large mathematical software libraries. If these libraries are expected to execute on several different computers, as they usually are, they should be portable. Unfortunately, portability is often in conflict with efficiency: portability dictates that code be common to several different computers while efficiency suggests tailoring code to the specifications of the individual computers.

The CRAY-1 computer is especially challenging in terms of reconciling portability with efficiency. Hardware vector arithmetic instructions must be utilized to reap the benefits of highly efficient code on the CRAY-1. Standard FORTRAN codes will compile and execute on the CRAY-1 with little or no modification. The FORTRAN compiler, CFT, does an admirable job of generating vector instructions for certain vector DO loops. In order to approach the maximum speed of the CRAY-1, it is necessary to rewrite some of the FORTRAN code and, often, to use the Cray Assembler Language (CAL). In order to approach the maximum speed of the computer for some problems, quite different algorithms must be considered. However, in this paper we assume that any required redesign has been accomplished.

If the code in question is modular, it is often fairly easy to identify those parts which should be specially coded. The CFT compiler can identify which subroutines are requiring most of the CPU time. But an astute programmer is needed to locate the inner loops which are CPU-intensive and to rewrite the FORTRAN code or replace it with calls to CAL-coded subroutines and functions. It is at this stage that the code starts to take on a flavor which is unique to the CRAY-1.

Boeing Computer Services offers mathematical software libraries which are portable, modular and efficient. The primary library, BCSLIB, is available on at least 6 different mainframe computers including the CRAY-1. Several additional libraries also are maintained. The authors are involved in the development of library modules which are specifically designed for the CRAY-1. This paper contains an overview of their experiences at tailoring mathematical software for the CRAY-1.

There are two fundamental ideas for making effective use of the CRAY-1 arithmetic hardware. First, the operands and results in the innermost loops should be structured into vectors and the computations should be vector-vector or vector-scalar operations such as adding two vectors or multiplying a vector by a scalar. Second, as many as possible of the CRAY-1's independent functional units should be brought to bear on the problem simultaneously. At the FORTRAN level, little can be done to promote a high level of concurrency. Frequently, astounding performance gains can be realized by using general purpose CAL-coded routines such as the BLAS, or by developing special-purpose CAL routines.

The first step in this project was to determine a priority of tasks based on two considerations: what basic numerical problems are most frequently encountered in large-scale scientific computing, and which problems are most amenable to CRAY-1 vectorization? Our first efforts were oriented toward linear algebra problems because they are often the innermost computations of many mathematical models. Furthermore, they are problems for which vector arithmetic instructions can be readily utilized.

We have implemented and evaluated several fundamental linear algebra subprograms, including the BLAS package, several versions of LINPACK and EISPACK, several versions of SPARSPAK, and we have worked with several application codes which have sparse matrix computations in their innermost parts. For example, we considered four versions of LINPACK: the standard FORTRAN version from Argonne National Laboratory, the version provided by Cray Research Inc. with FORTRAN and some CAL, the standard FORTRAN version but with calls to CAL BLAS provided by Cray Research Inc. (CRI), and the standard FORTRAN version with calls to CAL BLAS produced by one of the authors. A brief summary of our results follows.

o The CAL-coded versions of the real, single precision BLAS, supplied by CRI as part of their SCILIB scientific applications package, were enhanced, yielding increases in execution speed of 10% to 25%. For example our version of SNRM2 can execute at an asymptotic rate of 140 megaflops compared to the SCILIB version of SNRM2 which executes at an asymptotic rate of 113 megaflops. For comparison a CFT-compiled version of SNRM2 achieves an asymptotic rate of about 36 megaflops.

o The SCILIB version of LINPACK differs from the standard version in modifications of code which are better suited for the FORTRAN compiler and in the use of some CAL BLAS. The SCILIB version of SGEFA, an important LINPACK routine, executes in about one third the time required by the standard FORTRAN version. The version using locally developed BLAS outperforms the SCILIB version for matrices larger than 200x200.

o Minor modifications to EISPACK to aid the CFT vectorizer and to use the BLAS have dramatic effects on some eigenvalue paths. Throughout EISPACK, the execution rates are degraded considerably for two-dimensional matrices whose leading dimension is a multiple of eight. The degradation is due to memory bank conflicts.

o Speedups of 10% to 100% over standard FORTRAN codes can be achieved in sparse matrix factorizations even with little or no vectorization. We achieved such speedups both for problems held in envelope (variable band) format and also for more general and compact storage schemes. These improvements were achieved by replacing some of the innermost loops of key SPARSPAK routines with calls to CAL-coded routines.

In summary, the authors have found that significant speedups can be achieved by tailoring mathematical software to the CRAY-1's specifications. This can be done without affecting the user's program by changing only basic building block routines such as those in the BLAS, LINPACK, EISPACK, SPARSPAK and other frequently used packages. This approach also achieves a high degree of portability since the basic tools used all have portable FORTRAN equivalents.

FLEXIBILITY IN MATHEMATICAL SOFTWARE DEVELOPMENT USING OPTION ARRAYS

by R. J. Hanson
Sandia National Laboratories

F. T. Krogh
Jet Propulsion Laboratory

Introduction

Mathematical software (any type of software for that matter) has at least two goals. It should be easy to use. It should also be flexible and broad in its problem scope and thus satisfy a large number of the possible users of this software. These goals conflict with each other. It is the purpose of this paper to suggest some ways to reconcile this conflict by use of so-called "option arrays."

The methods we are proposing for the "option array" specifications are general. The implementation that we illustrate with an example is presented in FORTRAN, but the extension to other programming languages is obvious.

With any of the subprograms, say SUBPR (J), there will be M(J) options that the user can change. The author makes a numbered list of options for each subprogram, describing the features and usage of each of them.

An Example to Illustrate the Ideas

Solving dense systems of linear algebraic equations is a process that has received much attention from software specialists, Ref. [1]. To illustrate the techniques we propose we'll present the design of the options and a sample usage of a (mythical) subprogram for solving linear algebraic equations Ax=b, A = N by N real matrix. This design is for illustration only; a non-trivial real example is given in [2].

The nominal usage (no options) solves a system of equations with a single right-side vector. This usage involves the usual dimensioning of the required arrays, the definition of data, and the subprogram CALL statement.

```
Nominal Usage:
DIMENSION W(MDW,N+1), IOPT(1), ROPT(1),
*IWORK(N)
(Define matrix [A:b] within array W (*,*).)
IOPT(1)=99
CALL SL1 (W,MDW,N,IOPT,ROPT,IWORK)
(The solution vector, x, is returned in the
array W(*,N+1).)
```

This subroutine looks simple to use and narrow in scope. But now we have a (rare) user who wants to do a related computation:

1. There is no system to solve.
2. The determinant of A is desired.

The subprogram package author has provided a number of options in the subprograms that allow these related tasks to be done. The linear equation software involves subprograms SL1( ) and SL2( ). The subprogram SL1( ) calls SL2( ) to perform Gaussian elimination with partial pivoting. The subprogram SL1( ) is called by the user.

Option List for SL1 ( )	Option Number
Solve Ax=b with $k \geq 0$ right-side vectors; k=1 is nominal.	1
Solve (transpose of A)y=c with $m \geq 0$ right-side vectors.	2
Do not decompose the matrix A; this has already been done. Nominally the matrix A is decomposed each time SL1 ( ) is called.	3
Provide an option array to SL2 ( ); nominally no option array is provided to SL2 ( ).	4

Option List for SL2 ( )	Option Number
Provide column scaling for the matrix A.	1
Compute the determinant of A in the form det (A) = s * exp(t). Provide the parameters s and t as output values.	2
Don't redecompose the matrix A; this has already been done. Nominally the matrix A is decomposed each time SL2 ( ) is called.	3

We'll now give the values for the option array IOPT(*) that 1.) reset the number of right sides to zero, and 2.) compute the determinant of A in the form mentioned above. Comments in the right margin clarify the meaning of each entry. Note that the processing for the option arrays is terminated with the reserved option number, 99.

```
Optional Usage:
DIMENSION W(MDW,N), IOPT(8), ROPT(2), IWORK (N)
(Define matrix A within array W(*,*).)

IOPT(01) =  1  (Option number for SL1( ) to
                change number of right-side
                vectors.)
IOPT(02) =  0  (The number of right-side
                vectors.)
IOPT(03) =  4  (Option number for providing
                an option array to SL2( ).)
IOPT(04) =  6  (Pointer to start of option
                array for SL2( ).)
IOPT(05) = 99  (No more options for SL1( )
                remain.)
IOPT(06) =  2  (Option number for SL2( ) to
                compute the determinant of A.)
IOPT(07) =  1  (Store s in ROPT(1) and t in
                the following location.)
IOPT(08) = 99  (No more options for SL2 ( )
                remain).
CALL SL1 (W,MDW,N, IOPT,ROPT,IWORK)
```

The determinant of A is available in the form det(A)=ROPT(1)*EXP(ROPT(2)) after the return from subprogram SL1( ).

Ideas similar to those presented here are used in [2] and in some of the software in the libraries [3] and [4].

There have been a few important applications where the added flexibility provided by options within the software has saved expensive modifications to existing code. The effect of this has been to save the authors' time and the time of the user while a new programming effort was made. Another significant saving in time that prevented complications for several library users was realized by Krogh. The existing nonlinear least squares subprogram of Ref. [4] was modified to provide for simple bounds on the unknowns. This change was made using a new option number. Users of the previous version of the non-linear least squares subprogram continued to get the same results with the new version, and with the addition of the description for the new option, the old documentation still applied.

References

[1] Dongarra, J. J., Moler, C. B., Bunch, J. R., Stewart, G. W., LINPACK Users' Guide. SIAM Publications, Philadelphia, PA (1979).
[2] Krogh, F.T., Preliminary Usage Documentation for the Variable Order Integrators SODE and DODE. JPL Section 914 Computing Memorandum No. 399, Nov. 3, 1975.
[3] Haskell, K., Vandevender, W. Brief Instructions for Using the Sandia Mathematical Subroutine Library. (Vers. 8). SAND79-2382. (1980).
[4] JPL FORTRAN V Subprogram Directory. Ed. 5. JPL Doc. 1846-23 Rev. A. (1975).

(References [2] and [4] are internal JPL documents and are available from Krogh).

Hanson's contribution sponsored by the U.S. Departpartment of Energy under contract DE-AC04-76DP00789.

Krogh's contribution is one phase of research carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract No. NAS7-100, sponsored by the National Aeronautics and Space Administration.

Mechanizing the Maintenance of Source, Object, and Test Results — Or Why Should You Do All the Work?

Stuart Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

The Problem

The production and maintenance of complex computer programs involves an enormous amount of bookkeeping. First, there may be a multiple versions for different reasons:

> During development, errors will be corrected, facilities will be added or deleted, or there may be changes of implementation strategy.
>
> A program may be made up of almost independent parts, and different subsets of the collection may be provided to different users.
>
> The same functionality may be provided in more than one environment (different hardware, operating system, or language)

A particular version of a program is specified by a set of data and a set of processing steps to be performed. Some of the data will be fundamental (cannot be recreated automatically), such as program source text entered by humans and data produced by the real world. Other data can be derived, and sometimes stored, by reproducible processing steps applied to fundamental data and already derived data. The derived data may be used for other purposes such as debugging, producing listings, or running the program.

Keeping track of the processing steps and the state of the computation can be a major headache. Operations must be performed in a fixed order with complicated arguments and option specifications, errors must be considered, and records maintained. In simple cases, the processing is clear: "run the decks through the Fortran compiler and execute the resulting program". But such a description is often not applicable for significant jobs: libraries must to be established and updated, source files may be processed by a sequence of programs (macro processors, language translators), a single source file may participate in more than one compilation, and a single compilation may involve more than one source file (as in languages that permit "include files"). The processing steps are likely to be similar for different versions, but with small but essential differences. Remembering the sequence of operations to be performed is likely to be a significant task; keeping track of the state of intermediate files is even harder and more error-prone. If more than one person is involved in the construction or maintenance, they must communicate and avoid inconsistency in a disciplined way.

Testing and verification also beg to be organized. Simply making a change, running a simple test case, then installing the repaired version often succeeds, but is unsatisfactory once there are users who are remote or who depend on your program. It is important to maintain regression test suites, establish that all the program branches have been exercised, and check that the results are correct, or at least sufficiently accurate. The problem is aggravated if versions are being maintained for environments which are not immediately accessible.

Ameliorating Approaches

The ways to attack these problems may be viewed as a sequence of levels, at each of which the programmer surrenders some control to the computer in exchange for having the machine's increased assistance. All are based on an organized filing system. At first, this filing system might be a notebook full of penciled notations about bug fixes, changes for different versions, tape reel numbers, and job control sequences. It is amazing how far one can go with the manual approach, but eventually the notebook becomes illegible, disappears into a collaborator's office, or suffers some other dismal fate.

The obvious step is to store the notebook in a computer where it can be protected against decay. The program source will also probably be kept in computer files since text editors permit easy entry and modification. The notebook can contain file names instead of program descriptions. It is easy to capture frequent inputs (e.g., canned command sequences that can be issued without retyping) and outputs (e.g., successful test results to compare against new runs). If several people are working on the same project, the central file system can be used to coordinate and share the work and to communicate progress and problems. At this level, the computer is being used in a distant way: it stores data, but does not guide the work.

At the next level, one takes advantage of the structure of the computing system. If the file system permits long names or has a hierarchical organization, related forms of a file can be stored in files with computable names. If such a naming discipline is followed, production of versions can be automated by parametrizing some of the commands or by use of accessing functions. For example, Cargill showed how complicated sets of alternatives can be handled by disciplined use of the file system hierarchy.

The file system probably maintains certain useful information: names and lengths of files, perhaps their type (source, object, library, etc.), and the time the file was last changed. Such information is reliable, free, and can be extremely useful. For example, a program can make magically accurate deductions without explicit instruction: if the result of a compilation is needed, and if a source file that was edited since the last compilation was completed, then it is probably appropriate to recompile the source. A program can make this deduction and issue commands to the system with little intervention or thought by a human. (My Make program does this). This approach is independent of the details or content of the files, and only needs to know the use of the file, which is deducible from the name or attributes on many systems, and when it was last changed. At this stage of mechanization, the standard computer system is being used in a direct way; the user maintains his own files for his commands, versions of source, progress status, and so on, and invokes utilities explicitly.

If more help is needed or desired, the computer must be involved more intimately. The basic data are controlled by the computer, and may be stored in unreadable forms. Various versions may be intertwined for reasons of control or efficiency, and can only be examined through the tools. Thus, the SCCS system stores many versions of a text in a form that saves space, but explicit SCCS commands are needed to extract an old version or to store a new one. Tools at this level may also assist in managing the project by restricting access to programs, preventing multiple updates, and requiring explanations of the work done before accepting a new version. Note that these efforts have been based on a coarse unit of operation, the complete data file.

Finally, we reach the level of so-called program environments, which take over many of the operations done by programmers. In exchange for the convenience offered by the system, the programmer must accept its restrictions. This is an area of active research; some efforts in this area are Interlisp, Gandalf, Mentor, and Toolpack. The basic form of the fundamental data can be source text or a parsed representation. The division of responsibility among tools may then be radically different than in a conventional system. Some of the tools may be invoked invisibly. Editors, printers, and compilers make use of the single basic representation. An editor can check program syntax, and a compiler can be invoked automatically after an edit is complete. Maintaining statement usage counts and timing analyses can be implicit in the compilation process. Testing and debugging can be done in terms of the user's source language, without reference to machine-level objects. and test data might be saved semi-automatically. In exchange for these services, the user cannot apply standard tools to the data, since they are encoded and the environment must control all changes that are made.

A well-informed environment can control large objects such as libraries or executable programs, and can ensure that versions remain in parallel. The environment could produce instrumented forms of programs to monitor the testing: the instrumentation could record the statements that have been executed and the statements that consumed most of the machine time. The environment can also re-run regression tests before installing or distributing final copies, and maintain records on versions that have been sent out.

Applicability to Numerical Programs

The simpler approaches discussed above can all be applied to numerical programs on any flexible system. The more complete systems require some tailoring to the language and habits of the programmers. The program environments currently available are all research tools designed around languages that are rarely used for numerical programming. The problems of preprocessors, naming conventions, and enormous libraries require special consideration. The floating point domain presents some peculiar difficulties: a regression test may be required to achieve final results of satisfactory precision, but not necessarily to duplicate the previous run's output bit for bit, or even to produce comparable amounts of output. (Consider a change to an iterative algorithm which changes the number of iterations to produce satisfactory convergence).

Various projects are underway to attack some of these problems. In particular, the Toolpack project is designing a portable environment for Fortran-based programs.

NASA—JPL—Coml., L.A., Calif